

LABORATÓRIO DE MEIOS POROSOS E
PROPRIEDADES TERMOFÍSICAS
e
NÚCLEO DE PESQUISA EM CONSTRUÇÃO

Apostila de Programação Orientada a Objeto em C++



André Duarte Bueno, UFSC-LMPT-NPC
<http://www.lmpt.ufsc.br/~andre>
email: andre@lmpt.ufsc.br
Versão 0.4

22 de agosto de 2002

Apostila de Programação Orientada a Objeto em C++. Versão 0.4.

Distribuída na forma GPL (<http://www.gnu.org>).

Copyright (C) 2002 - **André Duarte Bueno**.

Esta apostila é “software” livre; você pode redistribuí-la e/ou modificá-la sob os termos da Licença Pública Geral GNU, conforme publicada pela Free Software Foundation; tanto a versão 2 da Licença como (a seu critério) qualquer versão mais nova.



Desenvolvida no Laboratório de Meios Porosos e Propriedades Termofísicas

(<http://www.lmpt.ufsc.br>)

e no Núcleo de Pesquisa em Construção (<http://www.npc.ufsc.br>),

com apoio do Curso de Pós-Graduação em Engenharia Mecânica

(<http://www.posmec.ufsc.br>) e da Universidade Federal de Santa Catarina

(<http://www.ufsc.br>).

Sumário

I	Filosofia de POO	37
1	Introdução a Programação Orientada a Objeto	39
1.1	Passado/Presente/Futuro	39
1.1.1	Passado	39
1.1.2	Presente	39
1.1.3	Futuro	40
1.2	Seleção da plataforma de programação	40
1.2.1	Seleção do ambiente gráfico - GDI (bibliotecas gráficas)	41
1.3	Ambientes de desenvolvimento	41
1.4	Exemplos de objetos	43
1.4.1	Um relógio	43
1.4.2	Um programa de integração numérica	44
2	Conceitos Básicos de POO	47
2.1	Abstração	47
2.2	Objeto (ou Instância)	47
2.3	Classes	48
2.4	Encapsulamento	48
2.5	Atributos (Propriedades/Variáveis)	49
2.6	Métodos (Serviços/Funções)	50
2.7	Herança (Hereditariedade)	50
2.7.1	Herança simples	51
2.7.2	Herança múltipla	51
2.8	Polimorfismo	52
3	Diagramas UML	53
3.1	Programas para desenho dos diagramas	53
3.2	Diagramas UML usando o programa dia	53
II	POO usando C++	57
4	Introdução ao C++	59
4.1	Um pouco de história	59
4.2	O que é o Ansi C++?	60
4.3	Quais as novidade e vantagens de C++?	60

4.4	Tipos de programação em C++	60
4.5	Compilar, linkar, debugar e profiler	61
4.6	Diferenças de nomenclatura (POO e C++)	62
4.7	Layout de um programa	62
4.7.1	Arquivo de projeto	62
4.7.2	Arquivo de cabeçalho da classe (*.h)	62
4.7.3	Arquivo de implementação da classe (*.cpp)	63
4.7.4	Arquivo de implementação da função main (programa.cpp)	63
4.8	Exemplo de um programa orientado a objeto em C++	64
5	Conceitos Básicos de C++	67
5.1	Sobre a sintaxe de C++	67
5.2	Conceitos básicos de C++	67
5.3	Palavras chaves do C++	70
5.4	Nome dos objetos (identificadores)	70
5.4.1	Convenção para nomes de objetos	70
5.5	Declarações	72
5.5.1	Sentenças para declarações	73
5.5.2	Exemplos de declarações ²	73
5.6	Definições	74
6	Tipos	79
6.1	Introdução ao conceito de tipos	79
6.2	Uso de tipos pré-definidos de C++	79
6.3	Uso de tipos do usuário	84
6.4	Uso de tipos definidos em bibliotecas externas (STL)	86
6.5	Vantagem da tipificação forte do C++	88
6.6	Sentenças para tipos	88
7	Namespace	91
7.1	O que é um namespace ?	91
7.2	Usando o espaço de nomes da biblioteca padrão de C++ (std)	91
7.3	Definindo um namespace ²	92
7.4	Compondo namespace ²	93
7.5	Sentenças para namespace	94
8	Classes	95
8.1	Protótipo para declarar e definir classes	95
8.2	Encapsulamento em C++ usando o especificador de acesso	96
8.3	Classes aninhadas ²	96
8.4	Sentenças para classes	97
9	Atributos	99
9.1	Protótipo para declarar e definir atributos	99
9.2	Atributos de objeto	99
9.3	Atributos de classe (estáticos)	100
9.3.1	Sentenças para atributos de classe	102

9.4	Atributos const	102
9.5	Atributos com mutable ²	103
9.6	Atributos com volatile ³	103
9.7	Inicialização dos atributos da classe nos construtores ²	103
9.8	Sentenças para atributos	104
10	Métodos	105
10.1	Protótipo para declarar e definir métodos	105
10.2	Declaração, definição e retorno de um métodos	106
10.2.1	Declaração de um método	106
10.2.2	Definição de um método	106
10.2.3	Retorno de um método	107
10.3	Passagem dos parâmetros por cópia, por referência e por ponteiro	107
10.3.1	Uso de argumentos pré-definidos (inicializadores)	110
10.3.2	Sentenças para declaração, definição e retorno de métodos	110
10.4	Métodos normais	111
10.5	Métodos const	114
10.6	Métodos estáticos	116
10.7	Métodos inline	120
10.8	Sentenças para métodos	124
11	Sobrecarga de Métodos	125
11.1	O que é a sobrecarga de métodos ?	125
11.2	Exemplos de sobrecarga	125
12	Uso de Ponteiros e Referências	127
12.1	Ponteiros	127
12.2	Criação e uso de objetos dinâmicos com ponteiros	129
12.2.1	Porque usar objetos dinâmicos ?	129
12.2.2	Controle da criação e deleção de objetos com ponteiros ²	131
12.3	Ponteiros const e ponteiros para const	131
12.4	Conversão de ponteiros ²	132
12.5	Ponteiro this	133
12.5.1	Sentenças para ponteiro this	133
12.6	Usando auto_ptr ²	133
12.7	Ponteiros para métodos e atributos da classe ³	136
12.8	Sentenças para ponteiros	136
12.9	Referências (&)	137
12.9.1	Diferenças entre referência e ponteiro	137
12.9.2	Referências para ponteiros ²	138
12.9.3	Sentenças para referências	138
13	Métodos Construtores e Destrutores	141
13.1	Protótipo para construtores e destrutores	141
13.2	Métodos construtores	141
13.2.1	Sentenças para construtores	142
13.3	Construtor default	143

13.3.1	Sentenças para construtor default	143
13.4	Construtor de cópia X(const X& obj)	143
13.5	Métodos destrutores	152
13.5.1	Sentenças para destrutores	152
13.5.2	Ordem de criação e destruição dos objetos	153
13.6	Sentenças para construtores e destrutores	154
14	Herança	155
14.1	Protótipo para herança	155
14.2	Especificador de herança	157
14.3	Chamando construtores da classe base explicitamente	159
14.4	Ambigüidade	160
14.4.1	Sentenças para herança	161
15	Herança Múltipla²	163
15.1	Protótipo para herança múltipla	163
15.2	Herança múltipla	163
15.3	Ambigüidade em herança múltipla	164
15.3.1	Herança múltipla com base comum	165
15.4	Herança múltipla virtual ²	166
15.4.1	Sentenças para herança múltipla	168
15.5	Ordem de criação e destruição dos objetos em heranças	169
15.5.1	Ordem de criação e destruição dos objetos em heranças virtuais	169
15.6	Redeclaração de método ou atributo na classe derivada	171
15.6.1	Sentenças para redeclarações	172
15.7	Exemplo de herança simples e herança múltipla	172
15.8	Análise dos erros emitidos pelo compilador ²	177
16	Polimorfismo²	179
16.1	Métodos não virtuais (normais, estáticos)	179
16.2	Métodos virtuais	181
16.2.1	Sentenças para métodos virtuais	181
16.3	Como implementar o polimorfismo	182
16.3.1	Sentenças para polimorfismo	185
16.4	Métodos virtuais puros (Classes abstratas) ² 10.1	186
16.4.1	Sentenças para métodos virtuais puros (classes abstratas)	186
16.5	Exemplo completo com polimorfismo	186
17	Friend	211
17.1	Introdução ao conteito de friend	211
17.2	Classes friend	211
17.3	Métodos friend	213
17.4	Sentenças para friend	214

18 Sobrecarga de Operador	215
18.1 Introdução a sobrecarga de operadores	215
18.2 Operadores que podem ser sobrecarregados	216
18.3 Sobrecarga de operador como função friend	216
18.4 Sobrecarga de operador como método membro da classe	217
18.5 Sentenças para sobrecarga de operador	218
18.6 Usar funções friend ou funções membro ?	219
18.7 Protótipos de sobrecarga	219
19 Implementando Associações em C++	227
19.1 Introdução as associações em C++	227
19.2 Associação sem atributo de ligação	227
19.3 Associação com atributo de ligação	228
20 Conversões	231
20.1 Protótipos para conversões	231
20.2 Necessidade de conversão	232
20.3 Construtor de conversão ²	233
20.4 Métodos de conversão (cast)	233
20.5 Conversão explícita nos construtores com explicit ²	234
20.6 Sentenças para construtor e métodos de conversão	235
20.7 Conversão dinâmica com dynamic_cast	236
20.7.1 Sentenças para cast dinâmico	238
20.8 Conversão estática com static_cast	239
20.9 Conversão com reinterpret_cast	239
20.10 Usando typeid	239
20.11 Verificação do tamanho de um objeto com sizeof	241
20.12 Referências e dynamic_cast	241
21 Excessões	243
21.1 Introdução as excessões	243
21.2 Conceitos básicos de excessões	244
21.2.1 try	246
21.2.2 throw	246
21.2.3 catch	246
21.3 Sequência de controle	247
21.3.1 Sequência de controle sem excessão	247
21.3.2 Sequência de controle com excessão	247
21.4 Como fica a pilha (heap) ²	248
21.5 Excessões não tratadas	250
21.6 Excessão para new	250
21.7 Excessões padrões	251
21.8 Sentenças para excessões	252

22	Templates ou Gabaritos	255
22.1	Introdução aos templates (gabaritos)	255
22.2	Protótipo para templates	255
22.2.1	Declaração explícita de função template	257
22.2.2	Sobrecarga de função template	257
22.2.3	Função template com objeto estático	257
22.3	Classes templates (ou tipos paramétricos)	257
22.4	Sentenças para templates	257
III	Classes Quase STL	259
23	Entrada e Saída com C++	261
23.1	Introdução a entrada e saída de dados no c++	261
23.1.1	Biblioteca de entrada e saída	261
23.2	O que é um locale ?	263
23.3	A classe <ios_base>	263
23.4	A classe <ios>	264
23.5	A classe <iomanip>	267
23.6	A classe <istream>	269
23.7	A classe <ostream>	272
23.8	A classe <sstream>	277
23.9	Sentenças para stream	278
24	Entrada e Saída com Arquivos de Disco	279
24.1	Introdução ao acesso a disco	279
24.2	A classe <fstream>	279
24.3	Armazenando e lendo objetos	282
24.4	Posicionando ponteiros de arquivos com seekg(), seekp(), tellg(), tellp() ²	286
24.5	Acessando a impressora e a saída auxiliar ³	287
24.6	Arquivos de disco binários ³	288
24.7	Executando e enviando comandos para um outro programa	288
24.8	Redirecionamento de entrada e saída	289
25	class <string>	293
25.1	Sentenças para strings	299
26	class <complex>	301
27	class <bitset>	305
IV	Introdução a STL	311
28	Introdução a Biblioteca Padrão de C++ (STL)	313
28.1	O que é a STL?	313

28.1.1	Características da STL:	313
28.1.2	Componentes da STL	314
28.2	Introdução aos containers	314
28.2.1	Tipos de Container's	314
28.2.2	Métodos comuns aos diversos container's	316
28.2.3	Typedef's para containers ²	317
28.3	Introdução aos iteradores (iterator's)	319
28.3.1	Tipos de iteradores	319
28.3.2	Operações comuns com iteradores ²	320
29	class <vector>	323
29.1	Sentenças para vector	328
30	class <list>	333
30.1	Sentenças para list	335
31	class <deque>	339
32	class <stack>	343
33	class <queue>	347
34	class <priority_queue>	349
35	class <set>	351
36	class <multiset>	355
37	class <map>	357
37.1	pair	357
37.2	map	357
37.3	Sentenças para map	359
38	class <multimap>	365
39	Algoritmos Genéricos	367
39.1	Introdução aos algoritmos genéricos	367
39.2	Classificação dos algoritmos genéricos	367
39.2.1	Classificação quanto a modificação do container	367
39.2.2	Classificação quando as operações realizadas	368
39.2.3	Classificação quanto a categoria dos iteradores	368
39.3	Funções genéricas	369
39.3.1	Preenchimento	369
39.3.2	Comparação	370
39.3.3	Remoção	370
39.3.4	Trocas	370
39.3.5	Misturar/Mesclar/Inverter	371
39.3.6	Pesquisa, ordenação	371

39.3.7	Classificação	373
39.3.8	Matemáticos	373
39.3.9	Operações matemáticas com conjuntos	374
39.3.10	Heapsort	374
39.3.11	Exemplos	375
40	Objetos Funções da STL	383
40.1	Introdução aos objetos funções da STL	383
40.1.1	Funções aritméticas	383
40.1.2	Funções de comparação	384
40.1.3	Funções lógicas	384
V	Programação Para Linux/Unix	387
41	Introdução a Programação GNU/Linux/Unix	389
41.1	O básico do GNU/Linux/Unix	389
41.1.1	Comandos do shell úteis	389
41.1.2	Expressões regulares ³	396
41.1.3	Programas telnet e ftp	398
41.2	Diretórios úteis para programadores	400
41.3	Programas úteis para programadores	400
42	Edição de Texto Emacs e VI	403
42.1	Comandos do editor emacs	403
42.1.1	Help	403
42.1.2	Movimento do cursor (use as setas de direção)	403
42.1.3	Cut/Copy/Paste/Undo	404
42.1.4	Arquivos	404
42.1.5	Pesquisa e substituição	405
42.1.6	Múltiplas janelas	405
42.1.7	Encerrando seção do Emacs	405
42.2	Comando do editor vi	406
43	Os programas diff, patch, indent	409
43.1	O programa diff	409
43.1.1	Sentenças para o diff	411
43.2	O programa patch	411
43.3	O programa indent	412
44	Compilando com gcc, g++	415
44.1	Protótipo e parâmetros do gcc/g++	415
44.2	Arquivos gerados pelo gcc/g++	416
44.3	Exemplo de uso do gcc/g++	416

45 Make	419
45.1 Um arquivo de projeto	419
45.2 Protótipo e parâmetros do make	420
45.3 Formato de um arquivo Makefile	420
45.3.1 Criando variáveis em um arquivo Makefile	420
45.3.2 Criando alvos em um arquivo Makefile	421
45.4 Exemplo de um arquivo Makefile	421
45.5 Sentenças para o make	423
46 Bibliotecas	425
46.1 Introdução a montagem de bibliotecas	425
46.1.1 ar	425
46.1.2 ranlib	426
46.1.3 nm	427
46.1.4 objdump	428
46.1.5 ldd	428
46.1.6 ldconfig	429
46.2 Convenção de nomes para bibliotecas	429
46.3 Bibliotecas usuais	429
46.4 Montando uma biblioteca estática (libNome.a)	429
46.4.1 Usando uma biblioteca estática	430
46.5 Montando uma biblioteca dinâmica (libNome.so)	431
46.5.1 Vantagens/desvantagens da biblioteca dinâmica	432
46.5.2 Usando uma biblioteca dinâmica	435
46.6 Sentenças para bibliotecas	435
47 Libtool	437
47.1 Introdução ao libtool	437
47.2 Forma de uso do libtool	437
47.3 Criando uma biblioteca sem o libtool	438
47.4 Criando uma biblioteca estática com o libtool	438
47.5 Criando uma biblioteca dinâmica com o libtool	439
47.6 Linkando executáveis	439
47.7 Instalando a biblioteca	439
47.8 Modos do libtool	440
47.9 Sentenças para o libtool	440
48 Debug (Depuradores, Debuggers)	443
48.1 Comandos do gbd	443
48.2 Exemplo de uso do gdb	443
48.3 Sentenças para o gdb	443
49 Profiler (gprof)	445
49.1 Sentenças para o profiler:	445

50	Versão de Depuração, Final e de Distribuição	447
50.1	Versão debug, release e de distribuição	447
50.1.1	Versão debug	447
50.1.2	Versão final (release)	447
50.1.3	Distribuição dos programas e bibliotecas	447
50.2	Sentenças para distribuição de código fonte	450
51	Documentação de Programas Usando Ferramentas Linux	451
51.1	Introdução a documentação de programas	451
51.2	Documentação embutida no código com JAVA_DOC	451
51.2.1	Exemplo de código documentado	451
51.2.2	Sentenças para documentação java_doc	452
51.3	Tutorial de configuração e uso do DOXYGEN	453
51.4	Exemplo de programa documentado	455
51.5	Exemplo de diagramas gerados pelo doxygen	457
51.6	Documentação profissional com sgml/xml (LyX)	458
52	Sequência de Montagem de Um Programa GNU	461
52.1	Introdução a programação multiplataforma com GNU	461
52.2	aclocal	463
52.3	ifnames	463
52.4	autoscan	463
52.4.1	Roteiro do autoscan	464
52.5	autoheader	464
52.5.1	Roteiro do autoheader	464
52.6	automake	465
52.6.1	Introdução ao automake	465
52.6.2	Sentenças para o automake	468
52.7	autoconf	468
52.7.1	Introdução ao autoconf	468
52.7.2	Protótipo do autoconf	469
52.7.3	Roteiro do autoconf	469
52.7.4	Estrutura de um arquivo configure.in	469
52.7.5	Exemplo de um arquivo configure.in	470
52.7.6	Macros do autoconf	470
52.7.7	Como aproveitar os resultados das pesquisas realizadas pelo autoconf	473
52.7.8	Variáveis definidas no arquivo configure.in e que serão substituídas no arquivo Makefile	475
52.8	autoreconf	476
52.9	./configure	477
52.10	Como incluir instruções do libtool em seu pacote gnu	477
52.10.1	Exemplo de arquivo makefile.am usando o libtool	477
52.10.2	Exemplo de arquivo configure.in usando o libtool	478
52.11	Exemplo Completo	478

53 Ambientes de Desenvolvimento no Linux	483
53.1 Kdevelop	483
53.1.1 O que é o kdevelop ?	483
53.1.2 Onde encontrar ?	483
53.1.3 Como instalar ?	483
54 Introdução ao Controle de Versões Com o CVS	485
54.1 O que é o CVS?	485
54.2 Comandos do cvs	486
54.3 Sequência de trabalho	488
54.3.1 Roteiro para criar um repositório	488
54.3.2 Para importar os arquivos de seu projeto antigo para dentro do repositório	489
54.3.3 Para baixar o projeto	490
54.3.4 Para criar módulos	491
54.3.5 Para adicionar/remover arquivos e diretórios	492
54.3.6 Para atualizar os arquivos locais	495
54.4 Versões, tag's e releases	496
54.4.1 Entendendo as versões	496
54.4.2 Para criar tag's	497
54.4.3 Para criar release's	498
54.4.4 Recuperando módulos e arquivos	499
54.5 Para verificar diferenças entre arquivos	500
54.6 <i>Verificando o estado do repositório</i>	501
54.6.1 <i>Histórico das alterações</i>	501
54.6.2 <i>Mensagens de log</i>	501
54.6.3 <i>Anotações</i>	502
54.6.4 Verificando o status dos arquivos	502
54.7 Ramos e Misturas (Branching and Merging)	503
54.7.1 Trabalhando com ramos	504
54.7.2 Mesclando 2 versões de um arquivo	504
54.7.3 Mesclando o ramo de trabalho com o ramo principal	505
54.8 Configuração do cvs no sistema cliente-servidor	505
54.8.1 Variáveis de ambiente	506
54.9 Frontends (cervisia)	506
54.10 Sentenças para o cvs	507
54.11 Um diagrama com os comandos do cvs	507
VI Modelagem Orientada a Objeto	509
55 Modelagem TMO (UML)	511
55.1 Modelo de objetos	511
55.1.1 Modelo de objetos->Ligações	511
55.1.2 Modelo de objetos->Associações	512
55.1.3 Modelo de objetos->Agregação	513
55.1.4 Modelo de objetos->Generalização e Herança	513

55.1.5	Modelo de objetos->Módulo / Assunto	514
55.2	Modelo dinâmico	515
55.2.1	Modelo dinâmico->Eventos ²	515
55.2.2	Modelo dinâmico->Cenários	515
55.2.3	Modelo dinâmico->Estados	516
55.2.4	Modelo dinâmico->Diagrama de Estados ²	516
55.2.5	Sugestões práticas	517
55.2.6	Relacionamento do modelo dinâmico com o modelo de objetos	517
55.3	Modelo funcional ³	518
56	Etapas de Desenvolvimento de Um Programa	519
56.1	Especificação	519
56.2	Análise orientada a objeto (AOO)	520
56.3	Modelagem de objetos	520
56.3.1	Identificação de assuntos	521
56.3.2	Identificação de classes	521
56.3.3	Identificação de objetos	522
56.3.4	Identificação de associações	522
56.3.5	Identificação de atributos	523
56.3.6	Identificação de heranças	523
56.3.7	Identificação de métodos (operações)	524
56.3.8	Teste dos caminhos de acesso	524
56.3.9	Iteração	525
56.3.10	Preparação do dicionário de dados	525
56.4	Modelagem dinâmica ²	525
56.4.1	Formação de interfaces	525
56.4.2	Preparação de um cenário	525
56.4.3	Identificação de eventos	526
56.4.4	Construa um diagrama de estados	526
56.4.5	Compare eventos entre objetos para verificar a consistência	526
56.5	Modelagem funcional ³	526
56.6	Projeto do sistema	527
56.6.1	Interface interativa ²	527
56.6.2	Simulação dinâmica	528
56.6.3	Identificação de subsistemas ²	528
56.6.4	Identificação de concorrências ²	528
56.6.5	Uso dos processadores ²	528
56.6.6	Identificação de depósitos de dados ²	528
56.6.7	Manipulação de recursos globais ²	529
56.6.8	Escolha da implementação de controle ²	529
56.6.9	Manipulação de condições extremas ²	529
56.6.10	Estabelecimento de prioridades	529
56.6.11	Estruturas arquitetônicas comuns ³	529
56.7	Projeto orientado a objeto ²	529
56.7.1	Implementação do controle	530
56.7.2	Métodos->localização	530

56.7.3	Métodos->otimização de desempenho	530
56.7.4	Ajustes nas heranças	530
56.7.5	Ajustes nas associações	530
56.7.6	Ajustes nos atributos de ligação	531
56.7.7	Empacotamento físico	531
56.7.8	O projeto de algoritmos	531
56.8	Implementação	531
56.9	Testes	532
56.10	Documentação de um programa	532
56.10.1	Documentação do sistema	532
56.10.2	Documentação dos assuntos	533
56.10.3	Documentação das classes	533
56.10.4	Documentação das relações	533
56.10.5	Documentação dos atributos	533
56.10.6	Documentação dos métodos	533
56.11	Manutenção	534
56.11.1	Extensibilidade, robustes, reusabilidade ²	534
Referências Bibliográficas		537
VII Apêndices		539
A Diretrizes de pré-processador		541
A.1	Introdução as diretrizes de pré processador	541
A.2	Compilação condicional	541
A.2.1	if	541
A.2.2	if...else	542
A.2.3	if...elif...elif...endif	542
A.2.4	define, ifdef, ifndef, undef	542
A.2.5	Macros	543
B Conceitos Úteis Para Programação em C/C++		545
B.1	Classes de armazenamento ²	545
B.2	Modificadores de acesso ²	546
B.3	Escopo das variáveis ²	549
B.4	Sentenças para classes de armazenamento, escopo e modificadores de acesso	551
C Operadores		553
C.1	Introdução aos operadores	553
C.2	Operadores de uso geral	554
C.2.1	Operadores aritiméticos (+, -, *, /, %)	554
C.2.2	Operadores de atribuição (=)	554
C.2.3	Operadores compostos (+=, -=, *=, /=)	554
C.2.4	Operadores relacionais (>, >=, <, <=, ==, !=)	554
C.2.5	Operadores lógicos (&&, , !, ==, !=)	555
C.2.6	Operador condicional (?)	555

C.2.7	Operador incremento (++) e decremento (--)	555
C.2.8	Operador vírgula (a,b)	556
C.2.9	Operador módulo (%)	556
C.3	Operadores de uso específico	557
C.3.1	Operador typedef	557
C.3.2	Operador sizeof e size_t	557
C.3.3	Operador de resolução de escopo (::)	557
C.3.4	Sentenças para operadores	557
D	Controle	561
D.1	if	561
D.2	if....else	561
D.3	if.....else if.....else if.....else	562
D.4	switch....case	562
D.5	expressão? ação_verdadeira : ação_falsa;	563
D.6	for(início;teste;incremento) ação;	563
D.7	while (teste){instrução};	564
D.8	do {ação} while (teste);	564
D.9	break	565
D.10	continue	565
E	Funções - Parte II	575
E.1	Uso de argumentos pré-definidos (inicializadores)	575
E.2	A função main() e a entrada na linha de comando ²	575
E.3	Funções recursivas ²	576
E.4	Uso de elipse ... em funções ³	578
E.5	Sentenças para funções	578
E.6	Exemplos	579
F	Ponteiros - Parte II	587
F.1	Operações com ponteiros (+/-) ²	587
F.2	Ponteiro void ²	587
F.2.1	Sentenças para ponteiro void	588
F.3	Ponteiro para ponteiro ³	588
F.4	Ponteiro de Função ³	589
F.5	Sentenças para ponteiros ²	590
G	Estruturas, Uniões e Enumerações	593
G.1	Estrutura (struct)	593
G.1.1	Definindo estruturas	593
G.1.2	Criando um objeto de uma estrutura	593
G.1.3	Acessando atributos de uma estrutura	594
G.1.4	Estruturas e funções ²	596
G.1.5	Lista encadeada ²	596
G.1.6	Estruturas aninhadas ²	596
G.1.7	Sentenças para estruturas	597
G.2	Uniões (union)	597

G.3 Enumerações (enumerated)	599
H Bibliotecas de C	601
H.1 <cmath>	601
I Portabilidade	603
J Bug / Debug	605
J.1 O que é um bug?	605
J.2 Uso de assert	605
J.3 Sentenças para evitar bugs	606
K Glossário	609
L Links Para Sites em C++	619
M Licença Pública Geral GNU	621
M.1 Introdução	621
M.2 Licença pública geral GNU termos e condições para cópia, distribuição e modificação	622
M.3 Como aplicar estes termos aos seus novos programas	626

Lista de Figuras

1	Por onde começar ?.	33
3.1	Tela do programa with class.	54
3.2	O programa DIA manipulando uma estrutura UML com representações de classes.	54
3.3	Representação de pacotes.	54
3.4	Representação de heranças, associações e agregações.	55
3.5	Representações diversas.	55
6.1	Tipos de dados e dimensões (sizeof).	80
9.1	Como fica o objeto b na memória.	100
9.2	Como fica o objeto na memória quando a classe tem atributos estáticos.	101
10.1	A classe TPessoa.	112
10.2	A classe TPonto.	121
12.1	Como declarar e usar um ponteiro.	128
13.1	A classe TAluno.	144
14.1	A herança entre TPonto e TCirculo.	156
15.1	Herança múltipla.	164
15.2	Como ficam os objetos b1, b2 e d em uma herança múltipla.	164
15.3	Herança múltipla com base comum.	165
15.4	Como ficam os objetos b1, b2 e d em uma herança múltipla com base comum.	166
15.5	Herança múltipla virtual.	167
15.6	Como ficam os objetos b1, b2 e d em uma herança múltipla com base B0 comum e virtual.	167
15.7	Seqüência de construção e destruição dos objetos em uma herança.	169
15.8	Hierarquia com herança múltipla normal e virtual.	170
15.9	Hierarquia de classes TPonto.	173
16.1	Ilustração da ligação dinâmica.	182
16.2	Hierarquia TPessoa, TAluno, TFuncionario, TAlunoFuncionario.	187
23.1	Esboço da biblioteca de manipulação de entrada e saída.	262
28.1	Métodos comuns aos diversos containers.	318
28.2	Métodos que retornam iteradores.	319

29.1	Métodos disponibilizados para vector.	327
30.1	Métodos disponibilizados para list.	334
31.1	Métodos disponibilizados para deque.	340
32.1	Métodos disponibilizados para stack.	343
35.1	Métodos disponibilizados para set.	351
36.1	Métodos disponibilizados para multiset.	355
37.1	Métodos disponibilizados para map.	358
42.1	O editor de texto emacs.	407
51.1	Ilustração da hierarquia TMatriz da biblioteca LIB_LMPT.	457
51.2	Ilustração da hierarquia da classe TRotulador3D da biblioteca LIB_LMPT.	458
51.3	Ilustração das dependências do arquivo TRotulador3D.	459
53.1	A tela do kdevelop (http://www.kdevelop.org).	484
54.1	Versões de um arquivo.	496
54.2	Criando um tag.	497
54.3	Criando um release.	499
54.4	Como ficam os ramos.	503
54.5	Um frontend para o cvs (o cervisia).	506
54.6	Diagrama com os comandos do cvs.	508

Lista de Tabelas

1	Arquivos da apostila no formato pdf.	31
4.1	Extensões usuais dos arquivos nas diferentes plataformas.	61
4.2	Diferenças na nomenclatura da POO e de C++.	62
5.1	Palavras chaves do <i>ANSI C++</i>	76
5.2	Convenção para nomes de objetos.	77
5.3	Exemplos de declarações.	77
6.1	Tipos e intervalos.	80
6.2	Diferenças de tamanho dos objetos padrões de C++ nas plataformas de 16 e 32 bits.	81
12.1	Conversão de ponteiros objetos.	132
14.1	Acesso herdado.	158
16.1	Métodos com ligação estática.	179
16.2	Métodos com ligação dinâmica.	180
18.1	Operadores que podem ser sobrecarregados.	216
23.1	Flags para o método setf.	264
23.2	Manipuladores da <iomanip>.	268
23.3	Caracteres de escape.	273
24.1	Modos de abertura do método open.	280
24.2	Modos de proteção do método open (atributos de arquivo).	280
24.3	Manipuladores para os métodos seekp e seekg.	286
28.1	Iteradores e posição.	317
41.1	Diretórios importantes para o programador.	400
41.2	Programas úteis para desenvolvedores de software no ambiente Linux.	401
46.1	Bibliotecas usuais.	430
48.1	Comandos do gdb.	444
52.1	Sequência para montagem de programa GNU.	462
52.2	Sequência executada pelo usuário.	462

C.1 Precedência dos operadores. 553

Listings

1	Exemplo de um listing.	32
4.1	Exemplo básico - Arquivo TAplicacao.h.	64
4.2	Exemplo básico - Arquivo TAplicacao.cpp.	64
4.3	Exemplo básico - Arquivo programa.cpp.	65
5.1	Usando saída para tela e nova linha.	70
5.2	Declaração de objetos e uso de cin e cout.	74
6.1	Tipos numéricos de C++.	80
6.2	Diferenças no uso de inteiro com sinal (signed) e sem sinal (unsigned).	82
6.3	Exemplo preliminar de definição de classe do usuário.	84
6.4	Exemplo preliminar de uso da classe vector da biblioteca STL	86
7.1	Definindo e usando um namespace.	92
10.1	Passando parâmetros por valor, referência e ponteiro.	108
10.2	Classe com atributo e método normal.	111
10.3	Classe com atributo e método const.	115
10.4	Classe com atributo e método estático.	117
10.5	Arquivo e87-TPonto.h.	121
10.6	Arquivo e87-TPonto.cpp.	122
10.7	Uso de métodos e atributos de uma classe.	123
12.1	Usando ponteiro para criar e usar objetos dinâmicos.	130
12.2	Comparando o uso de vetores estáticos de C, dinâmicos de C++, com auto_ptr de C++ e vector da stl.	134
12.3	Uso de referência.	137
13.1	Uso de construtor default e de copia.	144
13.2	Uso indevido do construtor de cópia em objetos com atributos dinâmicos.	150
14.1	Arquivo e87-TCirculo.h.	156
14.2	Arquivo e87-TCirculo.cpp.	157
14.3	Erro ao definir duas vezes o construtor default (Arquivo e101-ambiguidade.cpp).	160
15.1	Sequência de construção e destruição em herança múltipla virtual.	169
15.2	Arquivo e87-TElipse.h.	172
15.3	Arquivo e87-TElipse.cpp.	173
15.4	Arquivo e87-Programa.cpp.	174
15.5	Arquivo e87-TCirculoElipse.h.	176
15.6	Arquivo e87-TCirculoElipse.cpp.	176
15.7	Exemplo de mensagem de erro emitida pelo compilador g++ (no Linux) - Arquivo e87-TCirculoElipse.msg.	177
16.1	Exemplo de uso do polimorfismo.	183
16.2	Arquivo TPessoa.h.	186

16.3	Arquivo TPessoa.cpp.	188
16.4	Arquivo TAluno.h.	189
16.5	Arquivo TAluno.cpp.	191
16.6	Arquivo TFuncionario.h.	192
16.7	Arquivo TFuncionario.cpp.	194
16.8	Arquivo TAlunoFuncionario.h.	195
16.9	Arquivo TAlunoFuncionario.cpp.	196
16.10	Arquivo e91-class-Heranca.cpp.	197
16.11	Arquivo e92-class-Heranca-e-Polimorfismo.cpp.	201
16.12	Arquivo e93-class-Heranca-Multipla.cpp.	203
16.13	Arquivo makefile para exercícios e91, e92, e93.	207
16.14	Saída gerada pelo makefile dos exercícios e91, e92, e93.	208
17.1	Usando métodos e classes friend.	211
18.1	Arquivo e89-TPonto.h	220
18.2	Arquivo e89-TPonto.cpp.	221
18.3	Arquivo e89-Programa.cpp.	224
20.1	Uso do explicit.	234
20.2	Uso do dynamic-cast.	237
20.3	Uso de typeid.	239
21.1	Excessão: Divisão por zero	243
21.2	Excessão: Divisão por zero com controle simples.	243
21.3	Excessão: Divisão por zero com excessões.	245
21.4	Excessão e desempilhamento.	248
21.5	Excessão para new.	250
23.1	Formatação básica da saída de dados.	266
23.2	Formatação da saída de dados usando iomanip.	267
23.3	Formatação da saída de dados usando iomanip.	273
23.4	Uso de stringstream (ostream e istream).	277
24.1	Uso de stream de disco (ifstream e ofstream) para escrever e ler em arquivos de disco.	280
24.2	Leitura e gravação de objetos simples usando read/write.	282
24.3	Executando e enviando comandos para um outro programa (com ofstream).	288
24.4	Usando redirecionamento de arquivo.	290
25.1	Uso de string.	294
25.2	Uso de string e stringstream para executar um programa do shell.	298
26.1	Uso de complex.	302
27.1	Usando bitset - Exemplo 1	305
27.2	Usando bitset - Exemplo 2	308
27.3	Usando bitset com vector	309
29.1	Usando vector	329
30.1	Usando list.	336
31.1	Usando deque.	340
32.1	Usando stack.	344
35.1	Usando set.	353
37.1	Usando map.	360
39.1	Usando algoritmos genéricos.	375
39.2	Usando vector com algoritmo genéricos	378

40.1 Usando functional.	384
43.1 Arquivo e06a-hello.cpp.	409
43.2 Arquivo e06b-hello.cpp.	410
43.3 Arquivo diff.	410
43.4 Arquivo diff -c.	410
43.5 Arquivo diff -u.	411
43.6 Arquivo ex-vector-1-indent.cpp.	413
45.1 Arquivo makefile.	421
45.2 Exemplo de uso do programa make.	422
46.1 Saída do comando ar -help	426
46.2 Saída do comando nm -help	427
46.3 Saída do comando ldd /usr/bin/lyx.	428
46.4 Arquivo makefile com bibliotecas estáticas e dinâmicas.	432
46.5 Arquivo mostrando o uso do makefile.	434
47.1 Arquivo libtool -help.	437
51.1 Saída do comando doxygen -help.	455
51.2 Exemplo de código documentado no formato JAVA_DOC para uso com o programa doxygen.	455
54.1 Saída do comando: cvs -help-options	486
54.2 Saída do comando: cvs -help-commands	486
54.3 Saída do comando: cvs -help-synonyms	487
54.4 Saída do comando: cvs -import	489
54.5 Como fica o repositório após a importação	490
54.6 Saída do comando: cvs -H checkout	490
54.7 Saída do comando: cvs -H commit	492
54.8 Saída do comando cvs commit após adição de um módulo	492
54.9 Saída do comando: cvs -H update	495
54.10 Saída do comando: cvs -tag nome	497
54.11 Saída do comando: cvs commit -r 2	499
54.12 Saída do comando: cvs -diff	500
54.13 Saída do comando: cvs -log leiname.txt	501
54.14 Saída do comando: cvs -status leiname.txt	503
B.1 Modificadores de acesso.	546
B.2 Função e escopo - e14-escopo-a.cpp.	550
B.3 Função e escopo -e14-escopo-a.cpp.	550
C.1 Operadores de comparação.	558
C.2 Uso de sizeof.	558
D.1 Uso de for.	565
D.2 Uso de for encadeado.	566
D.3 Uso de while.	566
D.4 Uso de switch.	567
D.5 Uso de break.	569
D.6 Uso de continue.	569
D.7 Uso do operador de incremento.	570
D.8 Uso do operador while, exemplo 1.	571
D.9 Uso do operador while, exemplo 2.	572

D.10	Uso do operador módulo e do operador ?	572
E.1	Função recursiva.	576
E.2	Função cubo.	579
E.3	Função com void.	580
E.4	Função em linha (volume esfera).	581
E.5	Função em linha exemplo 2.	583
E.6	Exemplo de uso da biblioteca <cstdlib>.	583
F.1	Uso do operador de endereço e sizeof.	590
F.2	Uso de sizeof 1.	591
G.1	Uso de struct.	594
G.2	Uso de union.	598
G.3	Uso de union para apelidar atributo.	599
H.1	Uso de funções matemáticas.	601
	Lista de programas	

Prefácio

O desenvolvimento desta apostila teve como princípio a reunião dos conceitos de programação em C++ pelo próprio autor, uma espécie de resumo particular. O objetivo era organizar os conceitos de C++ e criar uma apostila de consulta rápida, em que a sintaxe de C++ seria apresentada de forma sucinta e direta.

Como o interesse pela programação orientada a objeto cresceu substancialmente, diversos alunos do LMPT¹ me solicitaram a realização de cursos rápidos abordando C++.

Com a apresentação destes cursos, identificava as maiores deficiências dos alunos, e, ao mesmo tempo ampliava a apostila.

Com o passar dos anos C++ evoluiu, centenas de novos conceitos foram adicionados. A medida que C++ evoluía, esta apostila também evoluía.

Para o desenvolvimento desta apostila foram consultados diversos livros de programação orientada a objeto, de C++, e de programação para Linux. As referências são classificadas a seguir.

- C++, [Bjarne, 1999, Margaret and Bjarne, 1993, Deitel and Deitel, 1997, Deitel and Deitel, 2001, Jeff and Keith, 1993, Steven and Group, 1993, Roberto and Fernando, 1994].
- UML, [Rumbaugh et al., 1994, Coad and Yourdon, 1993, Ann L. Winblad, 1993] [Martin and McClure, 1993].
- STL, [Eckel, 2000, Deitel and Deitel, 2001, Bjarne, 1999].
- LINUX, [Kurt Wall, 2001, Radajewski and Eadline, 1998, Vasudevan, 2001b, Vasudevan, 2001a, Dietz, 1998] [Cooper, 1999, Raymond, 2000, Cederqvist, 1993, ?] [Nolden and Kdevelop-Team, 1998, Manika, 1999, Gratti, 1999].
- Windows, [Ezzel, , Ezzel, 1991, Borland, 1996a, Borland, 1996b, Ezzel, 1993, Perry, 1995b] [Perry, 1995a, Schildt, 1990, Swan, 1994] [Wiener and Pinson, 1991, Steven and Group, 1993, Pappas and Murray, 1993].
- Processamento Paralelo, [Kurt Wall, 2001, Dietz, 1998, Hughs and Hughes, 1997] [Radajewski and Eadline, 1998].

Descreve-se a seguir as versões desenvolvidas.

¹LMPT significa Laboratório de Meios Porosos e Propriedades Termofísicas. Veja o site do LMPT, em <http://www.lmpt.ufsc.br>.

Versões

Versão 0.1: A versão 0.1 incluía apenas a programação orientada a objeto (Parte I) e a sintaxe de C++ (Parte II). Foi desenvolvida usando o editor word. Para gerar a versão (0.3) usei o staroffice² (5.2, free). Atualmente, me encantei com a beleza e facilidades do Lyx/latex, de forma que, esta e as próximas versões serão escritas usando L^AT_EX³.

Versão 0.2: Na versão 0.2 a apostila foi ampliada e mais detalhada. Foi acrescentada a programação para Windows usando a OWL (e o uso do Borland C++ 5.2) e a modelagem TMO (Parte VI).

Versão 0.3: Na versão 0.3 acrescentei a biblioteca padrão de C++, a STL (Parte IV) e a programação para Linux (Parte V).

Versão 0.4: Na versão 0.4 a apostila foi revisada e reestruturada. Foram adicionados exemplos externos (listagens de códigos externos devidamente testados⁴). A programação para Linux foi ampliada, bem como a parte relativa a STL. A programação para Windows usando a OWL⁵ e o Borland C++ 5.2 foi descontinuada. Na versão 0.4 diversos capítulos tiveram sua ordem invertida. Procurei trazer para o início da apostila os capítulos de programação orientada a objeto usando C++. Os capítulos de sintaxe, escopo, operadores, controles, ponteiros, funções e os relativos a programação em C, foram movidos para o apêndice. Pois boa parte de seus conceitos não se aplicam ao uso de classes, não sendo fundamentais em C++.

Atualizações (versões futuras):

Versão 0.5: Revisão por terceiros.

Versão 0.6: Inclusão de figuras, diagramas UML.

Versão 0.7: Unificação dos exemplos.

Versão 0.8: Inclusão de exemplo completo.

Versão 0.9: Inclusão de exemplos úteis. Informações adicionais sobre ambientes de desenvolvimento.

Versão 1.0: Revisão geral.

²Você pode obter cópias grátis do staroffice 5.2 em <http://www.staroffice.com/>. Atualmente pode-se obter o Open Office em <http://www.openoffice.org/>.

³Veja informações sobre tex em <http://biquinho.furg.br/tex-br/> e diversos links em <http://biquinho.furg.br/tex-br/links.html>. Veja informações sobre o LyX em <http://www.lyx.org/>.

⁴Testes realizados no Linux, RedHat 7x usando o compilador g++ da gnu. O Red Hat pode ser obtido em <http://www.redhat.com> e os programas da gnu em <http://www.gnu.org>.

⁵Observação importante. Um tempo enorme usado para aprender a usar a OWL foi literalmente perdido. Programas pagos como a OWL podem morrer de forma inesperada e deixar seus usuários orfãos. Este é mais um motivo para você usar programas livres.

Importante:

Esta é a versão 0.4 da apostila.

Considera-se que a mesma já pode ser publicada e usada por terceiros. Entretanto, deve-se ressaltar que se trata de uma versão beta, isto é, com deficiências e erros. Sugestões para atualização serão sempre bem vindas.

Se você encontrou erros na apostila, pode enviar um email para andre@lmpt.ufsc.br.

PS: No assunto do email inclua APOSTILA PROGRAMAÇÃO.

Sobre a Apostila

Esta apostila foi desenvolvida com o objetivo de concentrar os conhecimentos de *Programação Orientada a Objeto* e servir de base para um curso interno no Laboratório de Meios Porosos e Propriedades Termofísicas dos Materiais (LMPT) e no Núcleo de Pesquisa em Construção Civil (NPC).

O objetivo desta apostila é passar ao estudante, as noções de Programação Orientada a Objeto, de uma forma bastante rápida e direta, sendo desejável o acompanhamento dos estudos por um programador com experiência.

A apostila esta dividida nas seguintes partes:

1. **Filosofia de programação orientada a objeto (POO):** Se destina a transmitir os conceitos básicos de POO, a idéia, a filosofia e a nomenclatura utilizada.
Nesta parte descreve-se alguns exemplos de objetos, o que a POO representa em relação ao passado/presente e futuro da programação. Os mecanismos básicos e os conceitos chaves de POO.
2. **Programação orientada a objeto usando C++:** Apresenta a *programação orientada a objeto em C++*. Quais as características de um programa POO usando C++. Tipos padrões de C++, tipos do usuário e tipos da STL. Como declarar, definir e usar; classes, objetos, atributos e métodos. Como implementar a herança simples, a herança múltipla, o uso do polimorfismo, a sobrecarga de operadores, a conversão de tipos, os tipos genéricos (templates).
3. **Classes quase STL:** Apresenta-se um grupo de classes padrões de C++ e que não são exatamente classes da STL. Apresenta-se a entrada e saída de dados com as classes `<ios_base>`, `<istream>` e `<ostream>` e a classe `<sstream>`. Como realizar operações com arquivos de disco usando as classes `<fstream>`, `<ofstream>` e `<ifstream>`. A classe de strings padrões de C++ a `<string>`, a classe para tratar números complexos `<complex>`.
4. **Introdução a STL:** Apresenta-se a Standart Template Library (STL), que é uma biblioteca de objetos em C++. Descreve-se os conceitos básicos de containers e iteradores. Você vai aprender a usar um `vector<t>` para vetores, `list<t>` para listas duplamente encadeadas, `queue<t>` que representa uma fila, `stack<t>` que representa uma pilha (como em uma calculadora HP), uma `<deque>` que é uma fila com duas extremidades e classes para tratamento de conjunto de dados com chaves (`<set>`, `<multi_set>`, `<map>`, `<multi_map>`).

5. **Programação para Linux:** Descreve conceitos de *programação no mundo Linux*. Apresenta um resumo das ferramentas de programação do GNU/Linux, cobrindo g++, make, automake, autoconf, libtool, documentação com doxygen, controle de versões com CVS e programas como diff, patch, indent.
6. **Modelagem orientada a objeto:** Apresenta-se a modelagem orientada a objeto usando TMO. Mostra-se como montar o diagrama de uma *Análise Orientada a Objeto* (AOO) usando a modelagem TMO. A seguir apresenta-se as etapas de desenvolvimento de um software: a especificação, a análise orientada a objeto, o projeto do sistema, o projeto orientado a objeto, a implementação e teste; a manutenção e a documentação de um software.
7. **Apêndices: Conceitos gerais de programação em C/C++:** Descreve-se alguns conceitos gerais de programação em C/C++ como: diretrizes de pré-processador, classes de armazenamento e modificadores de acesso, funções, ponteiros, referências, estruturas, uniões.
8. **Exemplos de aplicações:** Apresenta-se um programa totalmente desenvolvido usando a programação orientada a objeto. São apresentadas todas as etapas de desenvolvimento, desde as especificações até o código em C++. Os arquivos com exemplos estão em dois formatos: o primeiro html⁶, permitindo uma visualização através de um browser. O segundo no formato ASCII (texto simples) com as extensões *.h (arquivos de declarações) e *.cpp (arquivos de implementação dos códigos).

Ao longo da apresentação dos temas, são incluídos exemplos. O aluno deve ler todos os tópicos e verificar o funcionamento com os exemplos. É importante compilar os programas e verificar o seu funcionamento.

Como fazer download da apostila

Os arquivos no formato pdf, podem ser baixados na home-page:

<http://www.lmpt.ufsc.br/~andre/>.

As listagens dos programas para GNU/Linux/Unix/Mac OS X, estão disponíveis em:

<http://www.lmpt.ufsc.br/~andre/ApostilaProgramacao/listagens.tar.gz>.

e para DOS/Windows em

<http://www.lmpt.ufsc.br/~andre/ApostilaProgramacao/listagens.zip>.

Dica: Crie um diretório apostila de programação em C++ e coloque ali os arquivos pdf e os exemplos descompactados.

⁶Verifique se já existem os arquivos desta versão da apostila, no formato html, no site www.lmpt.ufsc.br/~andre/.

Tabela 1: Arquivos da apostila no formato pdf.

Arquivo	Linux/Unix
<i>Filosofia de programação orientada a objeto</i>	P1-FilosofiaDePOO.pdf
<i>Programação orientada a objeto usando C++</i>	P2-POOUsoandoCpp.pdf
<i>Classes quase STL</i>	P3-ClassesQuaseSTL.pdf
<i>Introdução a STL</i>	P4-IntroducaoSTL.pdf
<i>Programação para Linux</i>	P5-Programacao-GNU-Linux.pdf
<i>Apêndices: Conceitos gerais de C/C++</i>	P6-Apendices.pdf
<i>Modelagem orientada a objeto</i>	P7-ModelagemOMT.pdf
Apostila completa	ApostilaProgramacao.pdf
Listagens de códigos	listagens.tar.gz

Como ler esta apostila

Para facilitar a leitura da apostila, alguns títulos tem um código informando a prioridade do mesmo. Este formato foi adotado por permitir a leitura da apostila por programadores iniciantes, intermediários e avançados.

Título (Iniciante)

O iniciante na programação em C++ NÃO deve ler os títulos (Título², Título³). Os títulos 2 e 3 podem incluir referências a conceitos que ainda não foram apresentados e só devem ser lidos por quem tiver experiência em C++ ou numa segunda leitura desta apostila.

Título² (Intermediário)

Se você já conhece C++ e quer se aperfeiçoar, leia também os títulos de nível 2.

Título³ (Avançado)

Se você já programa a algum tempo em C++ e quer aperfeiçoar seus conhecimentos, leia os títulos de nível 3. Os títulos de nível 3 abordam aspectos com os quais você vai se deparar depois de já ter feito alguns programas.

Também foram acrescentadas dicas gerais, dicas de performance e dicas para evitar bugs, utilizando-se os padrões abaixo.

Dica: Ao longo dos capítulos são apresentadas algumas dicas.

Performance: São dicas de como aumentar o desempenho de seus programas. As dicas de performance serão reunidas no *Capítulo Aumentando a Performance de Seus Programas*.

BUG: Cuidados para evitar a presença de bugs em seus programas. As dicas para evitar os bugs em seus programas estão sendo reunidas no *Capítulo Bugs*.

Para que o aluno possa fixar os conceitos, apresenta-se ao longo do texto protótipos de C++, exemplos e listagens de programas.

Protótipo: *Define a sintaxe de determinado comando, aparece em itálico.*

Exemplos:

- Exemplos textuais podem ser apresentados como ítems.

Exemplos:

```
/*Os exemplos não são programas completos,  
são pedaços de programas. Apenas ilustram  
determinada característica da linguagem e sua sintaxe.  
Os exemplos são apresentados em fonte fixa.*/
```

Exemplo:

```
int x = 2;
```

Listings: São exemplos de programas pequenos mas completos, na sua maioria foram testados. Cada programa é documentado, assim, o aluno vai entender o que cada linha está fazendo. Veja abaixo um exemplo de listing, nesta listagem apresenta-se um programa funcional, o tradicional “hello World”.

Listing 1: Exemplo de um listing.

```
#include <iostream>  
void main()  
{  
    std::cout << "Olá mundo!\n";  
}
```

Sentenças:

- São regras, exemplos e definições curtas e diretas.
- Se você encontrar termos desconhecidos dê uma olhada no glossário.
- ²Sentença de nível 2 só deve ser lida se você já conhece C++ e quer se aperfeiçoar.
- ³Sentença de nível 3 só deve ser lida por experts.

A Figura 1 mostra um diagrama onde você pode identificar a melhor sequência de leitura da apostila.

	Filosofia de POO	POO usando C++	Programação Linux
Introdução geral	1,2	3,4,5,6	39,40,42,43
Classes	53,54	7,8,9,10,12,17,19,I	
Herança	53,54	13,14,15	
Associação/gregação	53,54	18	
Ferramentas		11,16,21,H	41,47,48,49,52
Templates/Gabaritos		21	
Bibliotecas		22-38, J,	44,45,49,50
Entrada/saída		22,23	
STL		24,25,26-38,K	
Controle Erros/Debug		20,M	46,47,
Portabilidade			50,L
Diversos		A,B,C,D,E,F,G,N,O,P,Q	

Figura 1: Por onde começar ?.

Sobre o curso

Um curso rápido de programação orientada a objeto usando C++ pode seguir as aulas abaixo descritas. Cada aula deve ter pelo menos 2 horas. O aluno deve ler os capítulos da apostila e testar as listagens de código apresentadas. Evite baixar as listagens na internet, a digitação é importante para fixação da sintaxe de C++. As dúvidas principais serão esclarecidas em aula, dúvidas específicas serão atendidas fora da sala de aula.

1. Conceitos e filosofia de programação orientada a objeto. Objeto, classe, atributo, métodos, herança, associação, agregação. Abstração, encapsulamento, polimorfismo, identidade, mensagens.
2. POO usando C++. Introdução ao C++, conceitos básicos, palavras chaves, declaração, definição, tipos, namespace.
3. POO usando C++. Classes, atributos, métodos.
4. POO usando C++. Sobrecarga de operador, ponteiros, referência, construtor, destrutor.
5. POO usando C++. Herança, herança múltipla, polimorfismo, friend.
6. POO usando C++. Conversões, excessões, implementando associações e templates.
7. Quase STL. Entrada e saída para tela e disco. Classes string e complex.
8. STL, introdução a standart template library, conceitos básicos.
9. STL, containers e iteradores, a classe `<vector>`, exemplos.
10. STL, classes `<list>`, `<deque>`, `<queue>`, `<stack>`, `<map>`, `<multimap>`.
11. STL, iteradores, métodos genéricos, exemplos.

12. Conceitos gerais de programação em C/C++. Diretrizes de pré-processador, classe de armazenamento, escopo, matrizes, estrutura, união.
13. Apresentar o modelo de objetos: classe, assinatura, associação, agregação, herança múltipla, assunto. Apresentar o modelo dinâmico: Eventos, estados, cenários, diagrama de eventos, diagrama de estados.
14. Sequência de desenvolvimento de um software.
Exemplo: Desenvolvimento de uma biblioteca para manipulação de matrizes.
15. Programação para Linux. Introdução, emacs, diff, patch, indent, g++, make.
16. Programação para Linux. Desenvolvendo bibliotecas estáticas e dinâmicas, como debugar programas no Linux, o gnu profiler. Como distribuir versões de seus programas, como documentar seus programas (documentação de código e manuais).
17. Programação para Linux. Sequência de montagem de um programa GNU/Compliant.
18. Programação para Linux. CVS, controle de versões.
19. POO usando a OWL. Apresentar a biblioteca de classes da OWL. Criando programa em 20 passos. O que é a OWL, classes janela e aplicativo. Loop de mensagens.
20. POO usando a OWL. Tabela de resposta, menus, GDI, janelas e aplicativos MDI, barra de ferramentas.
21. POO usando a OWL. Quadros de diálogo e controles.

Obs: As aulas 19-21 podem ser orientadas para outra biblioteca.

Experiência do autor:

Ao longo dos últimos anos trabalhei no desenvolvimento dos programas:

Simulat: Programa de simulação de transferência de calor e umidade em telhas. Um programa para DOS, com acesso a tela, impressora, saída em disco e saída gráfica. O programa esta disponibilizado em www.lmpt.ufsc.br/andre/programas/simulan2000.exe.

Anaimp: Programa educacional de análise de imagens de meios porosos, escrito usando a biblioteca OWL (Object Windows Library, da Borland). Um programa para Windows, com janelas, ícones e etc. O programa não foi finalizado, uma versão alfa esta disponível em www.lmpt.ufsc.br/~andre/programas/Anaimp.zip.

Imago: Programa profissional de análise de imagens de meios porosos. Desenvolvido pela empresa ESSS (<http://www.esss.com>) com o comando do Eng. Marcos Cabral Damiani. Desenvolvi os sub-sistemas de determinação da permeabilidade pelo método do grafo de conexão serial e de determinação das configurações de equilíbrio. O programa esta disponibilizado em www.lmpt.ufsc.br/Imago.

LIB_LMPT: Uma biblioteca de sub-sistemas que cobre a área de análise de imagens (filtros, caracterização, reconstrução e simulação de processos em meios porosos reconstruídos).

Agradecimentos:

Gostaria de agradecer aos professores Paulo Cesar Philippi, Roberto Lamberts, Celso Peres Fernandes, José Antonio Bellini da Cunha Neto, Nathan Mendes, Fábio Santana Magnani, Saulo Guths, Vicente de Paulo Nicolau, Amir Antônio Martins de Oliveira Junior, Jean François Daian, que em algum momento e de alguma forma contribuíram para o desenvolvimento desta apostila.

Aos amigos Liang Zhirong, Luiz Orlando Emerich do Santos, Marcos Cabral Damiani.

Aos companheiros Aldomar Pedrini, Anastácio Silva, Fabiano Gilberto Wolf, Luís Adolfo Hegele Júnior, Paulo Cesar Facin, Rogério Vilain, Rodrigo Surmas, Carlos Enrique Pico Ortiz.

Aos alunos Adirley André Kramer, Carlos Eduardo Paghi, Diego Silva, Geziel Schaukoski de Oliveira, Henrique Cesar de Gaspari, Jaison Seberino Meiss, Luis Gustavo Bertezini, Saulo Guths, Rodrigo Hoffmann, Roberto Barazzeti Junior.

A UFSC, Universidade Federal de Santa Catarina, onde desenvolvi meus estudos.

Aos desenvolvedores do GNU/Linux e a idéia do software Livre.

Dedicatória:

Aos meus pais,

Bernardo Bueno e Alice Duarte Bueno.

Parte I
Filosofia de POO

Capítulo 1

Introdução a Programação Orientada a Objeto

Você verá neste capítulo o passado o presente e o futuro da programação, a seleção da técnica de programação e do ambiente gráfico. O que é a programação RAD. Exemplos de objetos e conceitos básicos de programação orientada a objeto. O que significa abstração, o que é uma classe, um objeto, um atributo. O conceito de herança e de polimorfismo.

1.1 Passado/Presente/Futuro

Vamos iniciar esta apostila falando um pouco de como se desenvolvia um programa e das coisas com as quais o programador precisava lidar, de seu universo. Depois descreve-se como se desenvolve um programa e finalmente vislumbra-se o que se espera do futuro.

1.1.1 Passado

As primeiras linguagens de programação eram bastante rústicas e obrigavam o programador a conhecer em excesso as características do hardware que estava usando. Um programa se dirigia para um equipamento específico e era extremamente complexo de desenvolver. Os programas eram desenvolvidos em linguagens de baixo nível como o assembler.

Com o passar dos anos, desenvolveram-se novas linguagens de programação, que iam desvinculando o programa do hardware.

Enquanto o desenvolvimento de hardware se dava a passos largos, o desenvolvimento de softwares estava atrasado cerca de 20 anos.

1.1.2 Presente

As linguagens de programação mais modernas permitem que um programa seja compilado e rodado em diferentes plataformas.

Mesmo com o surgimento de novas linguagens de programação, as equipes de programação sempre tiveram enormes problemas para o desenvolvimento de seus programas. Tendo sempre que partir do zero para o desenvolvimento de um novo programa, ou reaproveitando muito pouco dos códigos já desenvolvidos.

Programação estruturada

Com o desenvolvimento das técnicas estruturadas, os problemas diminuíram.

Na programação estruturada as funções trabalham sobre os dados, mas não tem uma ligação íntima com eles.

Programação orientada a objeto

Para tentar solucionar o problema do baixo reaproveitamento de código, tomou corpo a idéia da Programação Orientada a Objeto (POO). A POO não é nova, sua formulação inicial data de 1960. Porém, somente a partir dos anos 90 é que passou a ser usada. Hoje, todas as grandes empresas de desenvolvimento de programas tem desenvolvido os seus software's usando a programação orientada a objeto.

A programação orientada a objeto difere da programação estruturada.

Na programação orientada a objeto, funções e dados estão juntos, formando o objeto. Esta abordagem cria uma nova forma de analisar, projetar e desenvolver programas. De uma forma mais abstrata e genérica, que permite um maior reaproveitamento dos códigos e facilita a manutenção.

A programação orientada a objeto não é somente uma nova forma de programar é uma nova forma de pensar um problema, de forma abstrata, utilizando conceitos do mundo real e não conceitos computacionais. Os conceitos de objetos devem acompanhar todo o ciclo de desenvolvimento de um software.

A programação orientada a objeto também inclui uma nova notação e exige pôr parte do analista/programador o conhecimento desta notação (diagramas).

1.1.3 Futuro

Bibliotecas de objetos em áreas especializadas cuidadosamente desenhadas estarão disponíveis para dar suporte a programadores menos sofisticados. Os consumidores montarão seus programas unindo as bibliotecas externas com alguns objetos que criou, ou seja, poderão montar suas aplicações rapidamente contando com módulos pré fabricados.

O usuário final verá todos os ícones e janelas da tela como objetos e associará a sua alteração a manipulação destes objetos com as suas propriedades intrínsecas.

Exemplo, um ícone impressora representará a impressora de seu sistema computacional e permitirá a execução de uma impressão, a seleção do tamanho da página, entre outras operações com este objeto.

1.2 Seleção da plataforma de programação

Uma plataforma de computação envolve o hardware, o sistema operacional e a linguagem de programação. Pode-se desenvolver um programa para o PC usando DOS, para o PC usando WINDOWS, para o PC usando UNIX, para estações de trabalho usando UNIX, para MAC usando SYSTEM X, entre outros. Para desenvolver programas em um ambiente gráfico como o Windows, o Mac OS X, o Gnome ou o KDE, você vai ter de escolher:

- uma biblioteca gráfica e
- um ambiente de desenvolvimento.

Apresenta-se a seguir uma lista de bibliotecas gráficas que podem ser utilizadas para o desenvolvimento de programas com janelas. Depois apresenta-se rapidamente alguns ambientes de desenvolvimento.

1.2.1 Seleção do ambiente gráfico - GDI (bibliotecas gráficas)

A alguns anos desenvolvia-se um programa em computadores PC XT, PC AT, usando-se um ambiente em modo texto, não existiam janelas e ícones. Mais recentemente, praticamente todos os programas usam janelas, ícones, menus, . . . ; e são desenvolvidos para ambientes computacionais como o PC/Windows9X/NT/XP, estações de trabalho rodando UNIX, GNU/Linux com interface gráfica padrão MOTIF, ou mesmo MAC's rodando MAC OS System X.

Desenvolver um programa “For Windows”, assim que saiu o Windows 3.0 era uma verdadeira calamidade, pois o Windows só fornecia algumas funções básicas e o programador tinha que escrever praticamente tudo o que ia utilizar. O mesmo ocorria com os demais ambientes de janelas.

Hoje, desenvolver um programa para um ambiente de janelas ficou mais fácil, graças a bibliotecas de interfaces gráficas como a OWL, a VCL, a QT, entre outras, que fornecem toda uma hierarquia de classes e objetos que podem ser imediatamente herdados pelo seu programa. Você pode criar janelas, menus, botões, barras de ferramentas, entre outros objetos, com muita facilidade. Entretanto, para que você possa desenvolver um programa para um ambiente gráfico qualquer, você vai ter de saber programação orientada a objeto. Você só vai conseguir herdar e utilizar as bibliotecas fornecidas se compreender a programação orientada a objeto e a sintaxe de C++.

Em 2002, as bibliotecas mais utilizadas no ambiente Windows são a VCL do Builder e a MFC da Microsoft. No ambiente Linux as bibliotecas qt (da troll tech) e a biblioteca gtk (do GNU/gnome).

Dê preferência a bibliotecas multiplataforma.

1.3 Ambientes de desenvolvimento

Descreve-se a seguir alguns pacotes para desenvolvimento de programas em C++.

Windows

Em termos de ambientes de desenvolvimento, pode-se dizer que tanto o Borland C++ como o Visual C++ são programas bastante maduros e completos. Contam com geradores automáticos de código (como o AppExpert), em que o código é gerado a partir de algumas respostas fornecidas pelo programador. Contam também com ambientes de alteração das classes (como o ClassExpert). Espera-se que alguma versão futura do Builder C++ inclua as facilidades da montagem do programa usando um ambiente visual completo e inter-relacionado. Um programa GPL (software livre) muito bom é o DEVCC++.

- Microsoft Visual C++, ambiente completo com uso da biblioteca MFC (Microsoft Foundation Classes).
- Borland C++ 5, ambiente completo com uso da biblioteca OWL (Object Window Library).

- Borland C++ Builder, ambiente completo, tipo RAD¹ com uso da biblioteca VCL (Visual Class Library).
- *DevC++* (<http://www.bloodshed.net/dev/>), ambiente visual pequeno e simples de usar, usa as ferramentas da GNU.
- Sistema GNU, g++, make, automake, autoconf, libtool (<http://www.gnu.org>).

Mac

- Code warrior metroworks, ambiente completo com uso da biblioteca code warrior (?).
- Sistema GNU (g++, make, automake, autoconf, libtool).

Linux² (Unix)³

Os usuários novos de Linux/Unix/Mac OS X podem achar que o número de opções destas plataformas é reduzido, ledô engano. O número de ferramentas disponíveis é incrivelmente grande, lista-se a seguir, brevemente, alguns destes ambientes.

- kyllix <http://www.borland.com/kylix/index.html>. Ambiente com uso da biblioteca VCL (Visual Class Library).
- Code Warrior Metroworks, ambiente com uso da biblioteca code warrior (?).
- *kdevelop* <http://www.kdevelop.org/>, ambiente completo com uso da biblioteca qt ou kde.
- qt <http://www.trolltech.com>⁴, para o desenho de interfaces gráficas usando a biblioteca QT.
- glade <http://glade.gnome.org/> que utiliza o toolkit do gtk++ <http://www.gtk.org/>.
- dev C++ <http://www.bloodshed.net/dev/>, ambiente visual pequeno e simples de usar, usa as ferramentas da GNU..
- *Source navigator* <http://sources.redhat.com/sourcnav/>.
- Sistema GNU (g++, make, automake, autoconf, libtool). Pode-se desenvolver os programas com editores de texto simples e usar o make para compilação automatizada. Tem o cvs para controle de versões.

Observe que usar o sistema GNU garante uma maior portabilidade e uniformidade no desenvolvimento de seus programas, pois o mesmo está presente em praticamente todas as plataformas.

¹RAD= Aplicações de desenvolvimento rápido.

²A programação para LINUX é discutida na parte V desta apostila.

³Veja na Revista do Linux, edição 29, uma lista de ambientes de desenvolvimento para Linux (<http://www.revistadolinux.com.br/ed/029/assinantes/desenvolvimento.php3>).

⁴Veja uma pequena reportagem sobre o qt design na Revista do Linux, edição 31. <http://www.revistadolinux.com.br/ed/031/assinantes/programacao.php3>.

Programação visual (RAD): A programação visual não é necessariamente orientada a objetos. É normalmente mais fácil de programar a interface, entretanto, esconde do programador características vitais e deixa o código maior. Podem ser utilizadas para programas pequenos e médios (não se esqueça que todos os programas iniciam pequenos e depois se tornam grandes). Dentre os programas visuais atuais pode-se citar o *Visual Basic* (o mais difundido, mas não tem a mesma capacidade de outras linguagens), o *Delphi* (o segundo mais difundido, baseado no antigo Pascal), o *Builder* usa a linguagem C++ e é bastante completo, (<http://www.borland.com/cbuilder/index.html>). Para Linux tem o *Kylix* (<http://www.borland.com/kylix/index.html>).

Ferramentas CASE: Existem programas CASE para o desenvolvimento da análise orientada a objeto como o ood, o With Class (<http://www.microgold.com/index.html>) e o rational rose (<http://www.rational.com/>). Nestes o programador faz o diagrama das classes/ atributos /métodos e dos relacionamentos das classes. O programa conta com um módulo que gera o código do programa em diferentes linguagens a partir dos diagramas desenvolvidos, o que é uma grande vantagem.

Nenhum dos ambientes atuais é completo. Um ambiente de desenvolvimento completo (e ideal) teria três módulos. O primeiro módulo permitiria o desenvolvimento da análise (diagramas de análise), o segundo módulo permitiria a construção da interface visualmente e o terceiro módulo permitiria a alteração do código diretamente. Para que o sistema seja eficiente, o programador deve poder trabalhar em qualquer dos módulos e as correções serem feitas automaticamente em todos os arquivos.

Apresenta-se a seguir dois exemplos de objetos do mundo real e faz-se uma análise de algumas de suas características.

1.4 Exemplos de objetos

A programação orientada a objeto é baseada em uma série de conceitos chaves, que serão descritos no Capítulo 2. Vamos fazer uma análise de um objeto real e verificar que a programação orientada a objeto é baseada em conceitos que já conhecemos.

1.4.1 Um relógio

Retire o seu relógio do pulso e comece a analisá-lo. Verifique que o mesmo é um objeto real, que lhe dá algumas informações como hora, data, dia da semana, tem cronometro, alarmes; estas informações são atributos que são manipulados pelo relógio, **ou seja, um objeto tem atributos.**

O relógio também tem botões, como um botão de iluminação (noturna), um botão para selecionar o atributo a ser visto, um botão para acertar a hora. Podemos dizer que o acionamento destes botões corresponde ao acionamento de uma determinada função do relógio. **Logo, um objeto tem funções (métodos).**

Além dos botões, o relógio também tem uma caixa externa e uma pulseira, ou seja, um objeto relógio é formado de outros objetos. **Um objeto pode ser formado de outros objetos.**

Falamos de um relógio moderno, com alarmes e cronômetros; Mas um relógio antigo só informava a hora; De um relógio de bolso evoluiu-se para relógios de pulso, para relógios de parede, para relógios com alarmes, com cronômetros e assim pôr diante, ou seja, **um objeto pode evoluir de acordo com uma herança.**

Mas a informação principal do relógio é a hora certa, como um relógio não é uma máquina perfeita, ele pode atrasar. Neste caso, o dono do relógio usa a informação de um relógio padrão, com a hora certa, para acertar a hora. Neste exemplo, um objeto homem interagiu com o objeto relógio. **Podem existir interações entre os objetos.** Um atributo de um relógio, a hora certa, foi usada para acertar outro relógio, ou seja, **um objeto pode usar atributos de outros objetos.**

Você também sabe que existe uma fábrica de relógios, nesta fábrica estão as informações para se construir o relógio. **Vamos ver que uma classe é uma fábrica de objetos,** é na classe que se encontram as informações de como montar um objeto.

1.4.2 Um programa de integração numérica

Visão desorganizada: Precisa-se desenvolver um programa que realize a integração numérica da equação de uma parábola $y = a + b.x + c.x.x$.

O programador desorganizado imediatamente senta na frente do computador e começa a desenvolver o seu programa. Cria um arquivo único onde define as variáveis, a função e finalmente inclui o código para realizar a integração pelo método de simpson (porque é o que ele conhece e domina). Os nomes das variáveis são a1(o valor de y), a2 (o a da equação), a3 (o b), a4 (o c), a5 (ele não usa mas deixa definida). Define ainda s1, s2, s3, s4 (variáveis usadas no método de integração).

O programa vai funcionar, ele dará um nome como prog1 e armazenará no diretório diversos.

Depois de um mês ele já não lembra mais do nome do programa e onde o guardou e precisa agora desenvolver um programa de integração para uma outra função.

Bem, começa tudo de novo, pois não lembra o que significa a1, a2,...

Visão orientada a objeto: Todo o desenvolvimento do problema é feito de forma diferente. A intenção nunca é a de resolver um problema único e imediato.

O que quero é resolver uma integração numérica por qualquer método de uma equação genérica.

Ao olhar um livro de análise numérica descubro que existem um conjunto de métodos que podem ser utilizados para resolver o problema. As equações podem ser as mais diversas possíveis, mas tem algumas características em comum. A função parabólica obedece a forma $y = f(x)$.

Com relação aos métodos numéricos identifico os mais conhecidos Trapésio, Simpson, Gauss. Que tem em comum atributos como limiteInferior, limiteSuperior, numeroPontos, intervalo dx.

Assim, identifico alguns objetos, um objeto genérico de integração numérica, um objeto de integração por Simpson, outro por trapésio e outro por Gauss.

Identifico um objeto função da forma $y = f(x)$, que tem os atributos y, x e um método de cálculo que executa a função em sí.

O objeto integração deve receber o objeto função e poder realizar a integração desta função.

Diferenças em relação a visão desorganizada:

- Os objetos são representações de conceitos que já conheço.
- Os objetos, funções e variáveis tem nomes claros e precisos.
- Os objetos se relacionam da forma esperada, um programador iniciante terá uma visão facilitada do programa.

-
- O trabalho desenvolvido vai ser salvo como uma biblioteca de objetos, em um local adequado.
 - Todo o trabalho desenvolvido é documentado, facilitando o reaproveitamento dos códigos desenvolvidos.

Capítulo 2

Conceitos Básicos de POO

Neste capítulo vamos descrever cada mecanismo da programação orientada a objeto dando uma visão que você já conhece e uma visão associada a programação.

A *Análise Orientada a Objeto* (AOO) tem uma série de conceitos que auxiliam as pessoas a delinear claramente o problema e a identificar os objetos e seus relacionamentos.

Descreve-se a seguir os conceitos básicos da análise orientada a objeto, isto é, a abstração, o objeto, as classes, os atributos, os métodos, as heranças e o polimorfismo.

2.1 Abstração

No dicionário Aurélio, abstração significa considerar isoladamente coisas que estão unidas, ou seja, partimos do enfoque global de um determinado problema e procuramos separar os elementos fundamentais e colocá-los de uma forma mais próxima da solução. A idéia da abstração é identificar os elementos essenciais de um problema e suas propriedades fundamentais, separando ocorrências e atributos acidentais.

Para a análise orientada a objeto, abstração é o processo de identificação dos objetos e seus relacionamentos. A análise orientada a objeto permite ao programador concentrar-se no que um objeto é e faz sem se preocupar em como ele o faz. A abstração se dá em diferentes níveis: inicialmente abstrai-se o objeto; de um conjunto de objetos cria-se um conjunto de classes relacionadas, de um conjunto de classes cria-se uma biblioteca de classes.

2.2 Objeto (ou Instância)

Objetos são coisas do mundo real ou imaginário, que podemos de alguma forma identificar, como uma pedra, uma caneta, um copo, uma fada.

Um objeto tem determinadas propriedades que o caracterizam, e que são armazenadas no próprio objeto. As propriedades de um objeto são chamadas ainda de atributos.

O objeto interage com o meio e em função de excitações que sofre, realiza determinadas ações que alteram o seu estado (seus atributos). Os atributos de um objeto não são estáticos, eles sofrem alterações com o tempo.

Para a POO, um objeto é uma entidade única que reúne atributos e métodos, ou seja, reúne as propriedades do objeto e as reações as excitações que sofre.

Quando temos uma instância de uma classe, nós temos um objeto desta classe. Instância é um outro nome que se dá ao objeto, geralmente se refere a um objeto específico.

Identidade²: A identidade é uma propriedade que permite identificar univocamente um objeto. Os objetos se distinguem por sua própria existência, independente de seu conteúdo. Dois objetos são distintos mesmo que todos os seus atributos sejam iguais, ou seja, existe um único identificador para cada objeto.

Persistência²: É o tempo de vida de um objeto, podendo ser temporário ou permanente. Temporário quando só existe durante a execução do programa. Permanente quando é armazenado em um meio físico como a winchester. A vantagem dos objetos persistentes é que os mesmos podem ser acessados por mais de um programa, ou pelo mesmo programa em uma outra data, ou como um depósito de dados (banco de dados).

2.3 Classes

Quando falamos de classes, lembramos de classes sociais, de classes de animais (os vertebrados), de classes de objetos da natureza, de hierarquias. Ou seja, uma classe descreve um grupo de objetos com os mesmo atributos e comportamentos, além dos mesmos relacionamentos com outros objetos.

Para a análise orientada a objeto, uma classe é um conjunto de códigos de programação que incluem a definição dos atributos e dos métodos necessários para a criação de um ou mais objetos.

A classe contém toda a descrição da forma do objeto, é um molde para a criação do objeto, é uma matriz geradora de objetos, é uma fábrica de objetos. Uma classe também é um tipo definido pelo usuário.

Classificação²: Os objetos com a mesma estrutura de dados e com as mesmas operações são agrupados em uma classe. Um objeto contém uma referência implícita a sua classe, ele sabe a qual classe pertence.

Tipificação²: As classes representam os tipos de dados definidos pelo usuário. A tipificação é a capacidade do sistema distinguir as diferentes classes e resolver as conversões.

Modularidade²: A criação de módulos do programa que podem ser compilados separadamente. É usual separar a definição das classes de sua implementação.

Classes abstratas²: Uma classe é abstrata quando a mesma não é completa e não pode criar objetos (é como uma fábrica no papel). Uma classe abstrata pode surgir naturalmente ou através da migração de atributos e métodos para uma classe genérica. Somente classes **concretas** podem criar objetos.

2.4 Encapsulamento

Todos os equipamentos que utilizamos são altamente encapsulados. Tome como exemplo a sua televisão, ela tem um pequeno conjunto de botões que lhe permitem manipular os atributos do objeto televisor que são de seu interesse, como o canal, o volume, as cores.

Mas você sabe que o funcionamento do objeto televisor é extremamente complexo e que ao selecionar um novo canal, uma série de atributos internos são processados e alterados. Os atributos e funções internas estão encapsuladas, escondidas de você.

Para a análise orientada a objeto, encapsulamento é o ato de esconder do usuário informações que não são de seu interesse. O objeto atua como uma caixa preta, que realiza determinada operação mas o usuário não sabe, e não precisa saber, exatamente como. Ou seja, o encapsulamento envolve a separação dos elementos visíveis de um objeto dos invisíveis.

A vantagem do encapsulamento surge quando ocorre a necessidade de se modificar um programa existente. Por exemplo, você pode modificar todas as operações invisíveis de um objeto para melhorar o desempenho do mesmo sem se preocupar com o resto do programa. Como estes métodos não são acessíveis ao resto do sistema, eles podem ser modificados sem causar efeitos colaterais.

Exemplos:

- Um computador é um objeto extremamente complexo, mas para o usuário o que importa é o teclado, o monitor, o mouse e o gabinete.
- Ao utilizar um software como o StarOffice, a forma de uso é a mesma, seja num Pentium II-MMX ou num AMD K6. Os elementos invisíveis do computador (placa mãe, processador, memória) não alteram o uso do programa.
- As propriedades físicas de um determinado material de construção (telha) e os métodos de cálculo de suas propriedades (resistência a compressão, condutividade térmica...). Aqui, a telha é o objeto, as propriedades são seus atributos e o cálculo de suas propriedades são os métodos. Para o usuário o que interessa são as propriedades conhecidas, não interessa as equações, as variáveis intermediárias e a forma de cálculo, isto fica escondido.
- Num programa que calcule a área da curva normal, o cálculo interno pode ser realizado por um polinômio que aproxima a área ou através da integração numérica da equação da normal. A decisão de qual método de cálculo vai ser utilizado é realizada pelo objeto TNormal em função de um atributo interno o "limiteErro". O usuário externo cria o objeto TNormal informa o limite de erro e solicita o cálculo da área. O usuário não sabe qual método de cálculo vai ser utilizado, isto fica escondido.

2.5 Atributos (Propriedades/Variáveis)

A todo objeto podemos relacionar alguns atributos (propriedades). No exemplo do relógio a hora, a data. Na programação orientada a objeto, os atributos são definidos na classe e armazenados de forma individual ou coletiva pelos objetos.

Atributos de classe (coletivos): Quando um atributo é dividido entre todos os objetos criados, ele é armazenado na classe.

Exemplo: Um contador de relógios criados.

Atributos de objeto (individuais): Quando um atributo é individual ele é armazenado no objeto.

Exemplo: A hora de um relógio.

Cada relógio tem uma hora, que pode ou não estar certa.

2.6 Métodos (Serviços/Funções)

A todo objeto podemos relacionar determinados comportamentos, ações e reações.

As ações ou comportamento dos objetos são chamadas na análise orientada a objeto de métodos, assim, um método é uma função, um serviço fornecido pelo objeto.

Os comportamentos dos objetos são definidos na classe através dos métodos e servem para manipular e alterar os atributos do objeto (alteram o estado do objeto).

Exemplos:

- Um automóvel tem o comportamento de se locomover.
- Um computador de processar programas.
- Uma edificação de dar abrigo.
- Um meio poroso de permitir o fluxo de massa.
- Um equipamento de medição de realizar medidas.
- Uma método de conversão de uma imagem colorida em tons de cinza altera o estado da imagem, convertendo cada píxel colorido em um píxel cinza.

Mensagens²: Foi falado que um objeto tem determinados atributos (propriedades) e métodos (ações), e que o objeto reage ao meio que o envolve de acordo com as excitações que sofre. Em um programa orientado a objeto as excitações são representadas por mensagens que são enviadas a um objeto. Uma mensagem pode ser gerada pelo usuário, por exemplo, ao clicar o mouse.

Protocolo²: O protocolo é o conjunto de métodos que podem ser acessados pelo usuário, o conjunto de mensagens a que o objeto responde. Ou seja, o protocolo é o conjunto de métodos públicos da classe.

Ligação estática/dinâmica²: Ligação é o processo de identificar a posição dos métodos a serem executados. Na ligação estática o endereço dos métodos é definido durante a compilação do programa. Na ligação dinâmica o endereço dos métodos é definido somente durante a execução do programa.

2.7 Herança (Hereditariedade)

A herança esta relacionada as hierarquias e as relações entre os objetos.

No dia a dia, quando se fala de herança se refere a transferência de propriedades de um pai aos seus filhos, ou seja, aquilo que é do pai passa a ser do filho.

É comum ainda o dito popular “puxou o pai”, que significa que o filho tem as mesmas características do pai. De uma maneira geral as pessoas sabem que o filho puxou o pai mas não é ele,

ou seja não são a mesma pessoa. E que o filho apresenta determinadas características diferentes de seu pai.

Na análise orientada a objeto, herança é o mecanismo em que uma classe filha compartilha automaticamente todos os métodos e atributos de sua classe pai.

A herança permite implementar classes descendentes implementando os métodos e atributos que se diferenciam da classe pai.

Herança é a propriedade de podermos criar classes que se ampliam a partir de definições básicas. De classes mais simples e genéricas para classes mais complexas e específicas.

Exemplo:

- Um Pentium II tem todas as características do Pentium preservadas, mas acrescentou mais memória cache, a memória cache já existia mas foi ampliada.
- Uma placa mãe nova apresenta a interface USB, é uma novidade que antes não existia.

2.7.1 Herança simples

Quando uma classe herda as propriedades de uma única classe pai.

Exemplo:

- Herança genética, um menino herda as características genéticas de seus pais.

2.7.2 Herança múltipla

A herança múltipla ocorre quando uma classe tem mais de um pai.

Exemplo:

- Herança de comportamento, muitas vezes dizemos que um menino herdou o seu jeito engraçado do tio e estudioso do pai.

Nomes de classe²: Numa família os filhos e nétos compartilham os nomes de seus ancestrais, da mesma forma, em uma hierarquia de classes os nomes devem ser significativos, semelhantes e esclarecedores.

Superclasse²: Uma superclasse é a classe base de uma hierarquia de classes, é a classe mais alta na hierarquia (é a origem da árvore).

Compartilhamento²: As técnicas orientadas a objeto facilitam o compartilhamento de código através dos conceitos de herança. Além de um maior compartilhamento do código a análise orientada a objeto reduz a codificação em função da maior clareza dos diagramas desenvolvidos.

Cancelamento²: é a substituição de uma método da classe pai por outra na classe filho, pode ocorrer com os seguintes objetivos: cancelamento para extensão (ampliação das tarefas que eram realizadas), cancelamento para restrição (quando a tarefa não é mais necessária), cancelamento para otimização (quando se deseja aumentar a performance). Cancelamento por

conveniência (quando o cancelamento pode ser conveniente por um motivo qualquer, deve ser evitada pois é semanticamente errado). Os métodos não podem ser substituídos para terem um comportamento diferente do esperado.

2.8 Polimorfismo

A palavra polimorfismo significa muitas formas, e representa o fato de uma determinada característica (potência do motor do veículo) ser diferente para cada filho (tipo de veículo). Quem já andou de Volks e de Mercedes sabe bem a diferença.

Na natureza o conceito de polimorfismo é inerente ao processo de desenvolvimento, os seres evoluem, se modificam.

Exemplo:

- Num programa de simulação numérica pode-se ter a evolução dos métodos de integração numérica. Do método do Trapézio para o método de Simpson, para o método de Gauss.

Em suma, estamos partindo de um objeto mais simples e evoluindo. Mas os conceitos do objeto pai continuam a existir nos objetos descendentes, mesmo que tenham sofrido modificações, aperfeiçoamentos e assumido novas formas (polimorfismo).

O conceito de polimorfismo é fundamental para a análise orientada a objeto; sua aplicação se fundamenta no uso de uma superclasse, através do qual vamos desenvolver nossa hierarquia de classes.

Sinergia²: Os conceitos da análise orientada a objeto apresentam um efeito de sinergia (soma de qualidades), em que a soma dos diversos conceitos da AOO implicam num resultado mais positivo que o esperado.

A partir da versão 0.4 da apostila, o capítulo de *Modelagem TMO (UML)* e o capítulo *Etapas de Desenvolvimento de Um Programa* foram movidos para o final da apostila. Também foram movidos para o final da apostila tudo o que diz respeito a linguagem de programação C. Um curso de C++ é longo, e a experiência mostrou que iniciar com modelagem e depois abordar conceitos de C não funciona. Perde-se a relação de objeto conceitual (modelagem) com objeto da linguagem C++. O curso fica confuso. Como o número de conceitos novos é grande, ao chegar na parte interessante de C++, polimorfismo e STL, o aluno já não tinha mais capacidade de aprendizado.

Deve-se ressaltar que os seus primeiros programas usando POO consumirão o mesmo tempo que os desenvolvidos usando técnicas estruturadas. As vantagens do reaproveitamento aparecem a medida que os programas vão sendo desenvolvidos. Ou quando você já dispõe de uma biblioteca e pode desenvolver o programa a partir desta biblioteca.

Capítulo 3

Diagramas UML¹

Apresenta-se neste capítulo alguns diagramas UML. Os mesmos foram montados utilizando-se o programa dia, disponível em <http://www.gnome.org/gnome-office/dia.shtml>. Para aprender a usar em detalhes os diagrama UML consulte a página oficial da UML (<http://www.uml.org/>) e as referências UML, [Rumbaugh et al., 1994, Coad and Yourdon, 1993, Ann L. Winblad, 1993] [Martin and McClure, 1993].

3.1 Programas para desenho dos diagramas

Existem diversos programas para montagem dos diagramas UML, dentre os quais pode-se citar:

- dia (<http://www.gnome.org/gnome-office/dia.shtml>), é um programa pequeno, simples de usar e GPL. Faz parte do pacote office do gnome.
- rational rose (<http://www.rational.com>) é um pacote extremamente profissional, que além da montagem dos diagramas permite, simultaneamente, a implementação dos códigos. É um pacote pago, disponível para diversas plataformas.
- with class, outro pacote profissional e pago (<http://www.microgold.com/index.html>). Apresenta-se na Figura 3.1 a tela do programa with class.

3.2 Diagramas UML usando o programa dia

Não é objetivo desta apostila abordar o uso do programa dia, um pequeno manual do mesmo é obtido em <http://www.lysator.liu.se/~alla/dia/> e <http://www.togaware.com/linuxbook/dia.html>. O programa dia é um programa para montagem dos mais diversos diagramas. O programa contém um conjunto de componentes para montagem de diagramas UML. A tela do programa dia é ilustrada na Figura 3.2, observe que a lista de componentes UML esta selecionada. Observe no diagrama a direita, a representação de classes utilizando a notação UML.

Apresenta-se na Figura 3.3 a representação de pacotes utilizando a notação UML.

Apresenta-se na Figura 3.4 a representação de heranças utilizando a notação UML.

Apresenta-se na Figura 3.5 a representação de outros componentes da notação UML.

¹Esta é a primeira versão deste capítulo, posteriormente o mesmo será detalhado com os principais componentes de um diagrama UML.

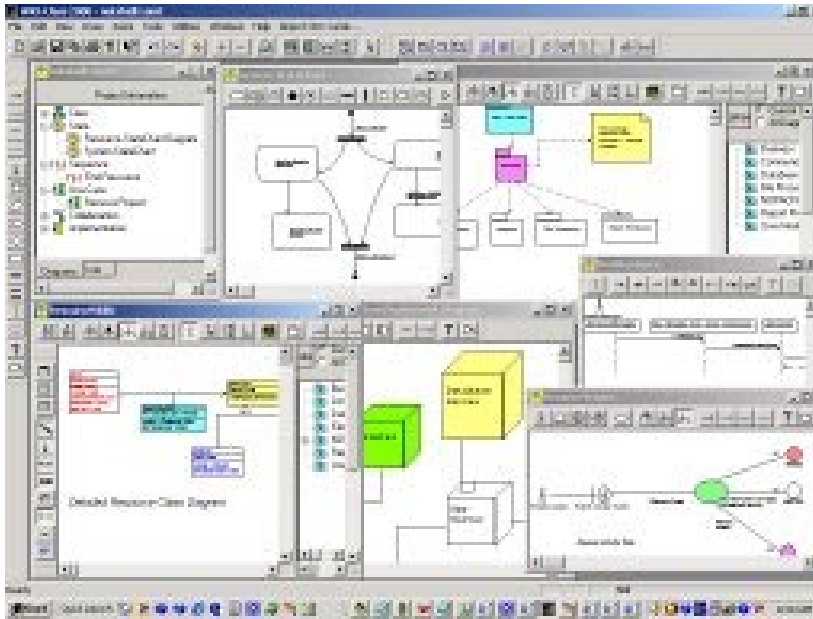


Figura 3.1: Tela do programa with class.

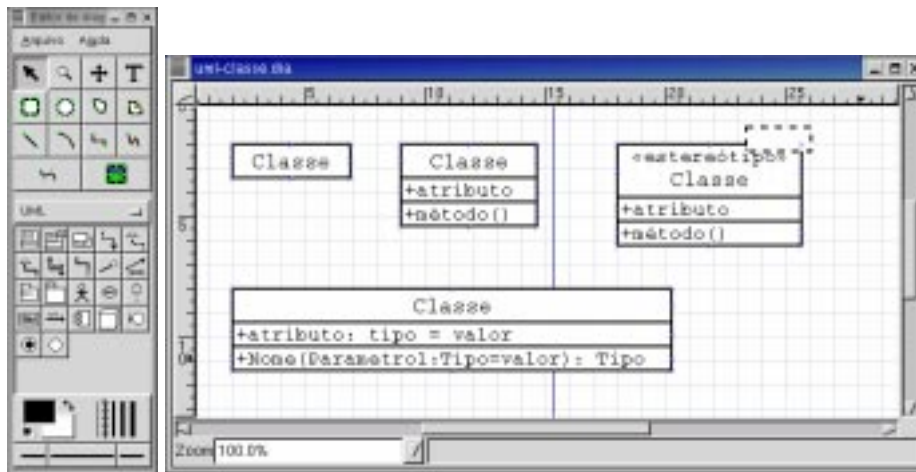


Figura 3.2: O programa DIA manipulando uma estrutura UML com representações de classes.



Figura 3.3: Representação de pacotes.

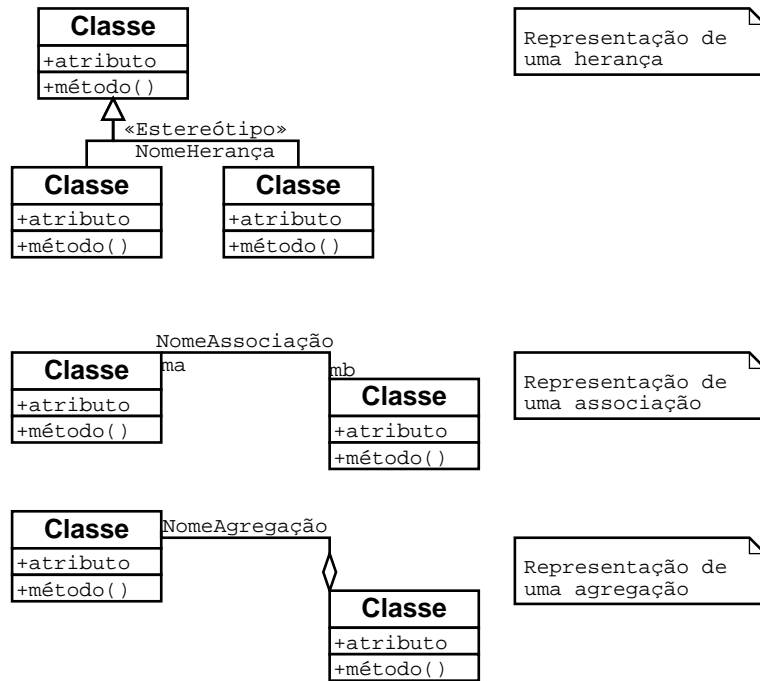


Figura 3.4: Representação de heranças, associações e agregações.

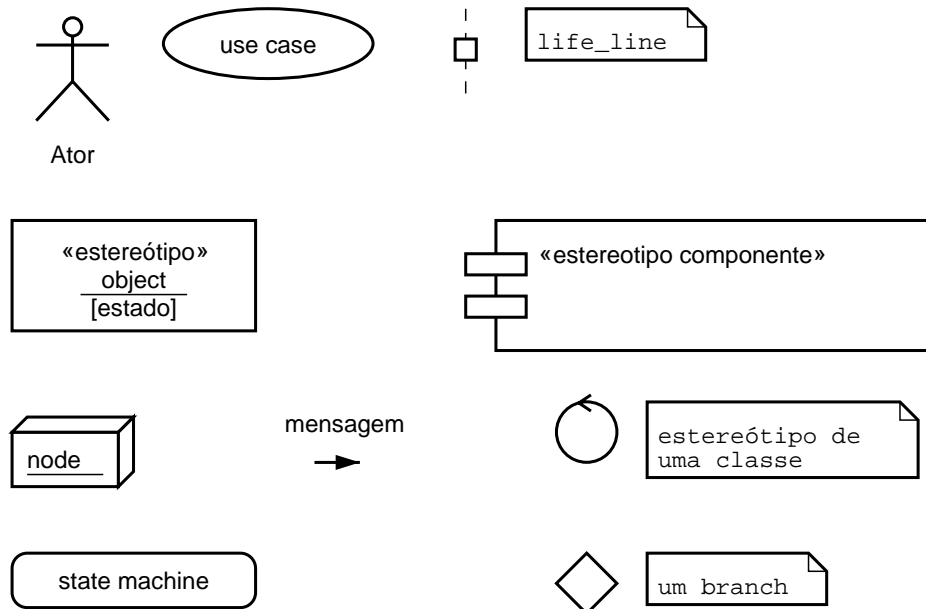


Figura 3.5: Representações diversas.

Parte II

POO usando C++

Capítulo 4

Introdução ao C++

Neste capítulo apresenta-se um pouco da história de C++, o que é o Ansi C++ e quais as vantagens de C++. Quais os tipos de programação em C++ ?. Quais os ambientes de desenvolvimento em C++, conceitos de compilação, linkagem, debuggers. As diferenças de nomenclatura entre a AOO e o C++. Como é o Layout de um programa em C++ e finalmente um pequeno exemplo.

4.1 Um pouco de história

C

A linguagem C teve origem na linguagem B (desenvolvida por Ken Thompson, em 1970), C foi desenvolvida por Denis Richard em 1972.

O ano de 1978 foi um ano histórico para a linguagem C, neste ano foi editado o livro "*The C Programming language*", que teve grande vendagem e foi o responsável pela divulgação de C. Este sucesso se deve ao fato de C ser independente de hardware. C tem sido utilizado em programas estruturados.

A linguagem C e o sistema operacional Unix foram desenvolvidos conjuntamente. Isto significa que C/C++ e ambientes operacionais como Unix, Linux e MacOS X tem uma interação muito íntima.

C++

Em 1980, Bjarne Stroustrup desenvolveu o C++ como um superconjunto de C e foi inicialmente chamado C com classes. Hoje em dia, quase todas as grandes empresas que desenvolvem softwares usam C++. Observe que o operador ++, é o operador de incremento, assim, C++ é o C incrementado.

O C++ apresenta uma série de vantagens em relação ao C, e se mostrou extremamente eficiente, nos mais variados campos de programação.

Mas afinal de contas devo aprender C e depois C++, ou ir direto para C++? O criador do C++, Bjarne Stroustrup afirma, "estou firmemente convencido de que é melhor ir direto para C++", [Bjarne, 1999].

4.2 O que é o Ansi C++?

O ANSI C++¹ é um comitê que estabelece os conceitos básicos da linguagem C++. Principalmente os referentes a sintaxe de C++. Se você desenvolver um programa compatível com o ANSI C++, pode ter certeza de que ele pode ser compilado por diferentes compiladores de C++ para diferentes plataformas.

Em 1990 foi aprovado o ANSI/ISO² 9899 que é o ANSI C. Em 1998 foi aprovado o ANSI/ISO C++.

4.3 Quais as novidades e vantagens de C++?

A linguagem C++ é uma das melhores linguagens de programação existentes por conseguir agrupar uma funcionalidade que envolve formulações altamente abstratas como classes, que permitem um trabalho de alto nível (trabalha-se a nível de conceitos) e formulações de baixo nível, como o uso de chamadas de interrupções que realizam tarefas altamente específicas.

Como novidades de C++ em relação ao C, podemos citar: O uso de classes, funções inline, conversão de tipo, verificação de argumentos de função, operadores para gerenciamento de memória (new/delete), referências, constantes, sobrecarga de operador, sobrecarga de funções, polimorfismo, templates (gabaritos), tratamento de exceções e espaços de nomes (namespace). Destes novos conceitos os que mais se destacam são o uso de classes, do polimorfismo e os gabaritos.

Como vantagens de C++ em relação ao C, podemos citar: Aumento da produtividade, maior reaproveitamento de código, maior qualidade geral do projeto, facilidade de extensão e manutenção. Maior compreensão geral por toda equipe de desenvolvimento.

Dica: Com C++ um único programador consegue gerenciar uma quantidade maior de código.

4.4 Tipos de programação em C++

Como visto, C++ é um superconjunto de C e foi desenvolvido para dar suporte a programação orientada a objeto. Qual a implicação dessa herança de C e C++ ?

Você vai se deparar com programas nos seguintes estados:

Programa estruturado escrito em C:

Uso os conceitos básicos de C, dados e funções separados.

Programa estruturado escrito em C++:

Usa alguns conceitos de C++ como cin/cout, switch, funções inline, const, referências. Ou seja, usa alguns acréscimos de C++.

¹ANSI = American National Standart Institute (Instituto Americano de Padrões e Medidas).

²ISO = Organização de Padrões Internacionais.

Programa baseado em objeto usando C++:

Usa os conceitos de classes e heranças. Inclue controle de acesso, funções friend. *Inclue o conceito fundamental de classes.*

Programa orientado a objeto em C++:

Inclue o conceito de polimorfismo, pode incluir o uso de conceitos da STL como containers e iteradores. *Inclue o conceito fundamental de polimorfismo.*

Programação genérica:

Inclue o uso de funções genéricas da STL, uso intensivo de containers e iteradores. *Inclue o conceito de código genérico.*

Esta apostila tem seu foco nos dois últimos tipos de programação.

4.5 Compilar, linkar, debugar e profiler

Descreve-se a seguir alguns conceitos gerais sobre programas.

Um programa: É composto de um ou mais arquivos encadeados. Um arquivo é composto por um conjunto de instruções de programação (em ASCII).

Fase de pré-processamento: É a primeira fase da compilação, verifica as instruções de compilação passadas com o sinal # ; Primeiro são incluídos os arquivos externos, depois são processadas as macros. O resultado da fase de pré-processamento é uma seqüência de símbolos que chamamos de unidade de tradução.

Compilador: O compilador encontra os erros de sintaxe do programa e realiza a tradução do código em linguagem de máquina. Depois de compilado o programa passa a ter um arquivo *.obj (*.o no Unix/Linux).

Linker: O linker transforma um ou mais arquivos *.obj (*.o) em um arquivo executável. Os arquivos que serão unidos são definidos em um arquivo de projeto ou em um arquivo makefile. Depois de linkado um programa tem um arquivo executável *.exe no Windows, a.out no Linux. Os erros de ligação são detectados pelo linker. Veja na Tabela 4.1 as extensões dos arquivos gerados nas diferentes plataformas.

Debugger: O debugger é um programa que ajuda o programador a encontrar os erros de programação, os famosos bug's.

Profiler: O profiler é um programa que ajuda a identificar os pontos do programa que consomem mais tempo (onde o programa está sendo lento); de posse dessa informação pode-se melhorar a qualidade do programa e a sua velocidade. Apenas rode o programa de dentro do profiler e analise os resultados de tempo de execução de cada função.

Tabela 4.1: Extensões usuais dos arquivos nas diferentes plataformas.

Situação	dos/Windows	Unix/Linux	Mac
antes de compilar	nome.h/nome.cpp	nome.h/nome.cpp	nome.h/nome.cpp
depois de compilar	nome.obj	nome.o	nome.o
depois de linkar	nome.exe	nome	nome

4.6 Diferenças de nomenclatura (POO e C++)

A Tabela 4.2 mostra as diferenças na nomenclatura da programação orientada a objeto e a nomenclatura de C++. Nesta apostila procurei usar sempre os nomes objeto, atributo e método. Mas em algumas ocasiões uso os os termos funções e variáveis. Vou chamar de função apenas as funções globais e funções de C. Vou chamar de método as funções que são implementadas como parte de uma classe. O objetivo é aproximar os conceitos da POO dos de programação em C++.

Tabela 4.2: Diferenças na nomenclatura da POO e de C++.

Nomenclatura POO	Nomenclatura C++
Objeto	Objeto
Classe	Classe
Método	Função/método
Atributo	Atributo, variável
Mensagem	Chamada de função
Subclasse	Classe derivada
Superclasse	Classe base
Hereditariedade	Derivação

4.7 Layout de um programa

O desenvolvimento de um programa inicia com a definição do arquivo de projeto, a seguir são criados os arquivos de cabeçalho (*.h) e os arquivos de implementação (*.cpp).

4.7.1 Arquivo de projeto

O arquivo de projeto define quais arquivos fazem parte do programa e em que sequência devem ser compilados, ou seja, contém uma lista com os nomes dos arquivos de cabeçalho (*.h) e de implementação (*.cpp) e a forma como os mesmos serão compilados. A organização dos programas separando o código em diversos arquivos facilita a manutenção do programa e possibilita um maior entendimento da estrutura dos programas. Todo processo de compilação/recompilação fica mais rápido.

Um arquivo de projeto tem a extensão *.ide ou *.prj (no Borland), *.mfc (no MFC), *.kdevelop (no kdevelop), e *.* (no Dev C++), podendo ser ainda um arquivo makefile³ (no Unix/Linux).

4.7.2 Arquivo de cabeçalho da classe (*.h)

A definição da classe é armazenada em arquivos de cabeçalho com a extensão *.h, veja o exemplo a seguir.

³Veja exemplo de arquivo makefile na seção 45.4.

```
//-----Arquivo TNomeClasse.h
/**
Cabeçalho do programa
Documentação geral da classe, o que é e representa
*/
//Declaração das bibliotecas standart's de C++
# include <iostream>
//Declaração da classe
class TAplicacao
{
///Declaração de atributo
tipo nomeAtributo;
///Declaração de método
tipo nomeFuncao(parametros);
//Controle de acesso
public:
};
```

4.7.3 Arquivo de implementação da classe (*.cpp)

As definições dos métodos das classes são armazenadas em arquivos de implementação com a extensão (*.cpp)⁴, veja o exemplo a seguir.

```
//-----Arquivo TNomeClasse.cpp
//Implementa as funções da classe TNomeClasse
//Bibliotecas de C++
#include <iostream>
//Bibliotecas do grupo de programadores
#include "TNomeClasse.h"
//Definição dos métodos da classe
tipo TNomeClasse::nomeFuncao(parametros)
{
Função em si;
}
```

4.7.4 Arquivo de implementação da função main (programa.cpp)

Você vai precisar de um arquivo com a definição da função main. É um arquivo com a extensão (*.cpp) e que usa as classes definidas pelo programador. Veja o exemplo a seguir.

```
//-----Arquivo programa.cpp
#include "TNomeClasse.h"
///Função principal
int main()
{
```

⁴Observe que arquivos de programação em C tem a extensão .c e de C++ a extensão .cpp (de C plus plus).

```

TAplicacao ap;
ap.Run();
return 0;
}

```

4.8 Exemplo de um programa orientado a objeto em C++

Apresenta-se a seguir um exemplo de um programa simples em C++. Neste exemplo estão presentes uma breve documentação da aplicação, da classe implementada e uma descrição dos atributos e métodos. Procurou-se incluir a maioria dos ítems presentes em um programa real. O exemplo está dividido em três arquivos. O arquivo e01-TAplicacao.h declara a classe TAplicacao e o arquivo TAplicacao.cpp define a classe TAplicacao. O arquivo programa.cpp inclui a função main. O programa inicia com a função main() e termina ao final desta função com um return(0).

Não se preocupe em entender o funcionamento do programa. Apenas preste atenção na divisão dos arquivos e no formato utilizado.

Listing 4.1: Exemplo básico - Arquivo TAplicacao.h.

```

//-----TAplicacao.h
//Declara uma classe minimalista
class TAplicacao
{
public:
    //Método de execução da aplicação
    void Run();
};

/*
Você verá posteriormente como declarar e definir classes.
Neste ponto não se preocupe com isto.
*/

```

Listing 4.2: Exemplo básico - Arquivo TAplicacao.cpp.

```

//-----TAplicacao.cpp
#include <iostream>

//Define método da classe.
#include "TAplicacao.h"

/**
O método Run escreve uma mensagem na tela
*/
void TAplicacao::Run()
{
    //inicio do método
    //std::cout escreve na tela
    //cout=c out
    std::cout << "Bem-vindo_ao_C++!"<<std::endl;
}

/*
Novidade:

```



```

- Inclusão de bibliotecas
#include <iostream>

-Saída para tela
    std::cout << "Bem vindo ao C++!";
*/

/*
Dica: Para compilar este arquivo no Linux, abra um terminal,
vá ao diretório com o arquivo e01-TAplicacao.cpp e execute
o comando abaixo:
    g++ -c e01-TAplicacao.cpp
*/

```

Listing 4.3: Exemplo básico - Arquivo programa.cpp.

```

//-----programa.cpp
//Inclue o arquivo "TAplicacao.h" que tem a definição do objeto TAplicacao
#include "TAplicacao.h"

//A função main, retorna um inteiro, se chama main e não tem nenhum parâmetro
int main()
{
    //Cria objeto do tipo TAplicacao
    TAplicacao ap;

    //Executa o método Run do objeto ap
    ap.Run();

    //A função deve retornar um inteiro o 0 indica que o programa terminou bem
    return 0;
}

/*
Novidade:
-A instrução #include <iostream> é usada para incluir o acesso
a biblioteca padrão de entrada e saída de dados do C++.

-A instrução #include "TAplicacao.h" é usada para incluir um arquivo,
procurando primeiro no diretório corrente e depois na path
do ambiente de desenvolvimento.

-0 objeto cout é usado para enviar caracteres para a tela.

-Dentro da função main a criação do objeto TAplicacao

-A execução do método Run do objeto TAplicacao

-0 retorno da função main
    return 0;
*/

/*
Dica: Para compilar o programa no Linux
    g++ programa.cpp TAplicacao.cpp

```

```
    Para executar o programa no Linux
    ./a.out
*/
/*
Saída:
-----
Bem vindo ao C++!
*/
```

Dica: C++ é extensivamente utilizado em aplicações científicas, em programas com interface gráfica e com muita interação com o usuário.

Dica²: Programas de engenharia, físicos, e matemáticos são bem representados em C++, pois as diferentes áreas da matemática são facilmente modeladas como classes em C++. Isto é, faz-se uma associação clara entre conceitos matemáticos e classes de C++.

Dica: Ao final de cada capítulo dedique cerca de 5 minutos para fazer uma revisão rápida dos conceitos apresentados.

Capítulo 5

Conceitos Básicos de C++

Apresenta-se neste capítulo alguns conceitos básicos de C++. As palavras chaves do C++, como você deve nomear seus objetos e como declarar e definir objetos.

5.1 Sobre a sintaxe de C++

Descrever todos os detalhes da sintaxe de C++ em programas orientados a objeto é um trabalho complexo e que levaria várias centenas de páginas. Para que esta apostila não ficasse muito grande, admite-se que o leitor tenha o acompanhamento de um programador para esclarecer suas dúvidas, ou que já tenha experiência em outras linguagens de programação.

A descrição da sintaxe de C++ segue dois modelos; No primeiro é descrita a sintaxe e apresentado ou um exemplo ou alguma informação extra. Na segunda é apresentada apenas a sintaxe, sem informações adicionais.

Por ser tão poderosa, a linguagem C++ tem um excesso de regras, o que ocasiona um aprendizado mais lento. Mas em C++ os programas tem um ganho de qualidade e versatilidade indiscutível em relação as linguagens mais simples como Basic, Pascal, Visual Basic e Delphi.

5.2 Conceitos básicos de C++

Apresenta-se a seguir alguns conceitos básicos de C++. Estes conceitos serão melhor compreendidos posteriormente, com as listagens de programas que são apresentadas.

Arquivo: É um texto contendo código fonte em C++ e comandos para o pré-processador.

Comentários: Um comentário em C usa os caracteres `/*` para iniciar o comentário e `*/` para encerrar o comentário.

Exemplo:
`/* comentário*/`

Um comentário em C++ usa duas barras (`//`).

Exemplo:

```
Aqui é programa ;           //Aqui é comentário.
//Todo o resto da linha passa a ser um comentário.
```

Símbolos: Existem cinco tipos de símbolos em um programa C++ (identificadores, palavras chave, operadores, literais e separadores).

Identificadores: Seqüência de letras definidas pelo programador (nome dos objetos, nome dos atributos e métodos).

```
Exemplo:
int x,y,z;           //x,y e z são identificadores
```

Palavras Chaves: São de uso interno do C++, tem significado para a linguagem, para o processo de compilação. Não podem ser usadas pelo usuário para nomear um objetivo.

Operadores: Símbolos cuja utilidade já é definida pelo C++, veja os operadores no Apêndice C, os operadores de C++ são:

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

```
Exemplo:
+ é o operador de soma.
* é o operador de multiplicação.
```

Literais¹: Tipos de variáveis previamente definidas pela linguagem, para representar objetos de uso corrente.

```
Exemplo:
int    x = 5;           //0 número 5 é um literal
char   c = 'a';        //a letra 'a' é um literal
float  y = 5.3;        //o número 5.3 é um literal
char*  nome = "joão";  //joão é um literal
```

Nome: Um nome denota um objeto, uma função, um enumerador, um tipo, um membro de classe, um modelo, um valor ou um label.

Atribuição: Quando se armazena algum valor no objeto.

Declaração: Diz que existe um objeto com nome fulano de tal, mas não cria o objeto. Uma declaração pode ser repetida.

```
Exemplo:
extern int a;
struct S;
extern const int c;           //não atribue valor
int função(); class Nome; struct s;
```

Definição: Cria um ou mais objetos e reserva memória. Uma definição não pode ser repetida.

```
Exemplo:
int b;
extern const int c = 1; //atribue valor
int função(){return 5;};
```

Classes de armazenamento: Define o tempo de vida de um objeto.

Pode ser estático ou dinâmico (registro, automático).

Escopo: Define onde um objeto é visível. Pode ser um objeto local, de função, de arquivo, de classe ou global.

Tipos fundamentais: Tipos de objetos previamente definidos pela linguagem:

```
Exemplo:
int x;                //objeto inteiro, com nome x
float y;              //objeto flutuante, com nome y
double z;             //objeto com dupla precisão, com nome z
long int xx;          //objeto inteiro longo com sinal, com nome xx
unsigned int xxx;     //objeto inteiro sem sinal, com nome xxx
```

Tipos derivados: Tipos definidos pelo programador como vetores, funções, ponteiros, referências, constantes, classes.

```
Exemplo:
int vetor1[50];      //vetor1 é um tipo do usuário
struct Pessoa        //Pessoa é um tipo do usuário
{char* nome;
int idade;};
```

Lvalues: Um objeto é uma região de armazenamento de memória. Um Lvalue é uma expressão que se refere a um objeto ou função (o retorno é algo ativo). Pode aparecer a esquerda do sinal igual (=), podendo ser alterado. Objetos especificados como const não são lvalues.

Blocos: Um bloco inicia com um “{” e termina com um “}”. Objetos criados dentro do bloco são objetos automáticos, os mesmos são automaticamente destruídos quando o bloco é encerrado. Objetos criados dentro do bloco não podem ser acessados externamente (escopo).

```
Exemplo:
int main()
{                //inicio do bloco
}                //fim do bloco
```

Diretrizes de pré-processamento: São informações/instruções que são passadas para o compilador com o símbolo #. Entenda o uso das diretrizes de pré-processamento na seção A na página 541.

Especificador: É uma palavra reservada da linguagem que é usada para atribuir determinadas propriedades aos tipos ou definir o tipo do objeto. Como exemplo um especificador inline, um especificador de tipo, um especificador de função e um especificador typedef.

Exemplo:

```
int x;                //int é o especificador
inline void f()      //inline é o especificador
    {cout <<"saida"<< endl;}
typedef float racional; //typedef é o especificador
```

5.3 Palavras chaves do C++

Uma linguagem de programação faz uso extensivo de determinadas palavras, denominadas palavras chaves. Estas palavras foram definidas para a linguagem C++, e são usadas pelo programador com algum objetivo específico. Como estas palavras já tem um significado pré definido para a linguagem, você não pode declarar um objeto com o mesmo nome de uma palavra chave, pois o compilador faria uma confusão e acusaria erro.

Com o objetivo de economizar nas palavras, algumas palavras chaves tem mais de uma utilidade. Como exemplo a palavra chave virtual, a mesma pode especificar uma função virtual ou uma herança virtual. Outro exemplo é void, que para ponteiros é um ponteiro para qualquer coisa, e para funções significa ausência de parâmetros ou ausência de retorno. Lista-se as palavras chaves do ANSI C++ na Tabela 5.1. As palavras chaves estão classificadas de acordo com seu uso.

5.4 Nome dos objetos (identificadores)

O nome de um objeto deve começar com uma letra (a-z, A-Z, ou underscore _).

A linguagem C++ difere maiúsculas e minúsculas, ou seja, AA é diferente de aa.

Caracteres válidos: a-z A-Z 0-9 +-*/=,,:;\ " ' ~ !# \$ & ()[]{} ^ @

Caracteres inválidos: ++ - == & & // << >> >= <= += -= *= /= ?: :: /**

5.4.1 Convenção para nomes de objetos

Para facilitar a leitura do programa, estabelece-se uma convenção de como os objetos devem ser nomeados, veja na Tabela 5.2 uma convenção para nomes de objetos.

Apresenta-se a seguir um outro exemplo, observe ao final a seção novidades e a seção saída. A seção novidades descreve alguma coisa nova que foi utilizada. A seção saída mostra a saída gerada pelo programa.

Listing 5.1: Usando saída para tela e nova linha.

```
#include <iostream>

int main()
{
    //'\n' coloca uma quebra de linha
    std::cout << "Welcome\n";
}
```

```

    // '\a' emite um beep
    std::cout << "to C++!\n\a";

    // '\t' coloca um tab
    std::cout << "Bem" << '\t' << "Vindo!";

    // cada \n coloca uma quebra de linha
    std::cout << "\nB\nne\nnm\n\nV\nni\nnn\nd\nno\n\na\nno\n\nC++\n!\n";

    return 0;
}

/*
Novidades:
-----
Uso de \a para emitir um beep,
    std::cout << "to C++!\n\a";

Uso de \t para acrescentar um tab,
    std::cout << "Bem" << '\t' << "Vindo!";

Observe que usa "Welcome" com aspas duplas para uma palavra ou frase
e '\t' aspas simples para um único caractere.
*/
/*
Saída:
-----
Welcome
to C++!
Bem      Vindo!
B
e
m

V
i
n
d
o

a
o

C++
!
*/

```

O programa inicia incluindo a biblioteca padrão de C++, a `<iostream>`, na linha **#include <iostream>**. A `<iostream>` é uma biblioteca usada para entrada e saída de dados. A `<iostream>` fornece o objeto `std::cout`, o mesmo é usado para enviar uma mensagem para a tela. Observe o uso do operador `<<`, que indica, envie estes caracteres para saída (cout = C out).

O exemplo inclui ainda a função `int main()`. A função `main` é a função inicial de um programa em C++.

Dentro da função `main`, envia para a tela a mensagem "Welcome\n". O caracter `\n` é usado

para colocar uma quebra de linha depois da palavra “Welcome”. A seguir envia para a tela a mensagem “ to C++” e “Bem \tVindo” O caracter \t é usado para colocar uma tabulação entre a palavra “Bem” e a palavra “Vindo”. Na penúltima linha envia para a tela a mensagem “Bem Vindo ao C++”, incluindo, entre cada caracter uma nova linha '\n'. A última linha da função main é a **return 0;**, a mesma é usada para finalizar a função main e o programa, retornando para o sistema operacional o valor 0, que indica sucesso na execução do programa. Observe no final da listagem como ficou a saída.

Apresenta-se a seguir como declarar e definir objetos em C++.

5.5 Declarações

Uma declaração introduz um ou mais nomes em um programa e especifica como esses nomes devem ser interpretados. Uma declaração não reserva memória para o objeto, apenas diz que ele existe. Uma declaração tem dois componentes os especificadores² e os declaradores³.

Protótipo: *Especificador Declarador;*
Especificador, especifica o tipo do objeto.
Declarador, é o nome do objeto.

Exemplo:

```
class Point;      //introduz um nome de classe
typedef int I;   //introduz um sinônimo para int
int x;           //introduz um objeto do tipo int com nome x
```

Declaração simples: Consiste em declarar um objeto de cada vez.

Exemplo:

```
int x;
```

Declaração múltipla: Consiste em declarar vários objeto de uma única vez.

Exemplo:

```
float r,s,t;
```

Declaração com inicialização: Consiste em declarar um objeto e ao mesmo tempo atribuir um valor a este objeto.

Exemplo:

```
int u = 7;
float x = 5.2;
```

²Especificadores: Os especificadores indicam o tipo fundamental, a classe de armazenamento ou outras propriedades dos objetos declarados.

³Declaradores: Os declaradores especificam o nome dos objetos e opcionalmente modificam o tipo com um *. Um declarador pode especificar um valor inicial para o identificador que esta sendo declarado (=).

5.5.1 Sentenças para declarações

- Sempre coloque uma linha em branco antes de uma declaração.
- Sempre declarar um objeto por linha.

```
Exemplo:  
//cria int* a; e int b;  
int *a, b;  
//use o formato abaixo, é mais claro  
int* a;  
int b;
```

- Sempre que possível iniciar os objetos na sua declaração.
- Sempre usar espaços para maior clareza.
- Sempre usar indentação⁴.
- Coloque parenteses extras para aumentar a clareza do código.
- Use nomes curtos para objetos muito usados e nomes longos para objetos e métodos pouco usados.
- No C temos que declarar todas as variáveis no início do programa, no C++ podemos declarar os objetos em qualquer parte. O ideal é declarar os objetos perto de onde os utilizaremos.
- Um objeto só pode ser usado depois de ter sido declarado.
- Objetos podem ser modificados com as palavras chaves `const` (constante, não muda), `volatile` (podem mudar de forma inesperada), `static` (duram toda a execução do programa).
- Objetos estáticos são inicializados com 0.
- Objetos locais, criados dentro de um bloco (ou função) não são inicializados. Você precisa passar um valor para os mesmos.
- Você não deve confundir a classe de armazenamento com o escopo do objeto. A classe de armazenamento se refere ao tempo de vida do objeto (temporário ou permanente), já o escopo do objeto define onde ele pode ser utilizado (onde é visível). Veja uma descrição dos conceitos de classes de armazenamento, escopo das variáveis e modificadores de acesso na seção B.2.
- ²Use a palavra chave `export` para informar que aquele objeto/classe/método é acessível externamente.

5.5.2 Exemplos de declarações²

Apresenta-se na Tabela 5.3 exemplos de declarações de objetos. Inclui declaração de matrizes, funções e ponteiros.

⁴No Linux para deixar o código organizado você pode usar o programa `indent` (Veja seção 43.3).

5.6 Definições

Uma definição faz com que seja reservada a quantidade adequada de memória para o objeto e seja feita qualquer inicialização apropriada.

Uma declaração de um objeto é uma definição, a menos que contenha um extern e não tenha um inicializador.

Apresenta-se a seguir um exemplo com declaração de objetos e entrada e saída de dados. Novamente, o programa inicia incluindo a biblioteca padrão de C++ para entrada e saída de dados, a <iostream>. Como dito anteriormente, a <iostream> fornece o objeto std::cout usado para escrever na tela. A <iostream> fornece ainda o objeto std::cin, que é usado para entrada de dados. Observe o uso do operador >> que indica, armazene a entrada do usuário neste objeto.

Listing 5.2: Declaração de objetos e uso de cin e cout.

```
#include <iostream>

int main()
{
    //um int é um tipo pré-definido, serve para armazenar inteiros
    //Uma declaração envolve
    //Tipo_do_objeto Nome_do_objeto;
    //Na linha abaixo o tipo é int, o nome a
    int a;

    //escreve na tela "Entre com a:"
    std::cout << "Entre com a:";

    //Espera que o usuário digite o valor de a e um enter.
    //armazena o valor digitado no objeto a
    std::cin >> a;

    //A vírgula pode ser usada para separar objetos que estão sendo declarados
    int b,c;

    std::cout << "Entre com b:";
    std::cin >> b;

    //Observe que os objetos são declarados perto de onde eles começam
    //a ser usados. Abaixo declara variável do tipo int, com nome soma
    int soma;

    //verifica o tipo de a, o tipo de b, se compatíveis realiza a soma
    //e então armazena o resultado em soma
    soma = a + b;

    //escreve na tela o resultado de soma
    std::cout << "Soma=" << soma ;

    //o comando endl (usado abaixo)
    //envia para a iostream cout um final de linha (linha nova)
    //e descarrega o bufer armazenado em cout.
    //Isto significa que o endl obriga o programa a escrever na tela
    imediatamente.
    std::cout << endl;
```

```
    return 0;
}
/*
Novidade:
-----
-Declaração de objetos
-Entrada de dados
-Saída de dados
-Realização de conta

O objeto cin, declarado no arquivo <iostream>
é usado para armazenar dados digitados no teclado
em um objeto do usuário. cin é usada na linha:
    std::cin >> a;
*/
/*
Saída:
-----
[andre@mercurio Cap2-Sintaxe]$ ./a.out
Entre com a:5
Entre com b:6
Soma =11
*/
```

Dica: Os programas foram compilados e rodados em um PC usando Linux. O compilador utilizado é o g++, o compilador da GNU. Por default, o compilador da GNU gera um executável com nome a.out. Para compilar o programa abra um terminal, vá para o diretório onde o programa esta localizado e digite **g++ nomePrograma.cpp**. Para executar o programa digita-se **./a.out**. Para compilar e executar o programa no ambiente Windows consulte os manuais de seu ambiente de desenvolvimento.

Tabela 5.1: Palavras chaves do *ANSI C++*.

Tipos

char	double	enum	float	int	long	short
------	--------	------	-------	-----	------	-------

Modificadores de tipos

auto	const	extern	register	signed
static	typedef	unsigned	volatile	static

Controle

break	case	continue	default	do
else	for	goto	if	return
switch	while			

Lógicos

and	and_eq	bitand	bitor	not
not_eq	xor	xor_eq	or	or_eq

Memória

new	delete
-----	--------

Controle de acesso

public	private	protected
--------	---------	-----------

Conversões

const_cast	dynamic_cast	static_cast	reinterpret_cast
------------	--------------	-------------	------------------

Excessões

try	throw	catch
-----	-------	-------

Diversos

asm	class	explicit	friend	namespace
operator	register	typename	typeid	this
using	struct	sizeof	union	void

Tabela 5.2: Convenção para nomes de objetos.

Tipo de objeto	Formato do nome
Variáveis constantes:	CONTADOR;
Nome de classes:	TNomeClasse;
Nome de métodos/funções:	Minúsculas();
Atributos:	minúsculas;
Atributos estáticos:	\$minúsculas;
Nome de classes derivadas:	Deve lembrar a classe base.

Tabela 5.3: Exemplos de declarações.

Sintaxe da declaração	Tipo efetivo	Exemplo
tipo nome[];	matriz do tipo	int count[];
tipo nome[3];	matriz do tipo c/ 3 elementos	int count[3]; //0,1,2
tipo* nome;	ponteiro para tipo	int* count;
tipo* nome[];	matriz de ponteiros para tipo	int* count[];
tipo* (nome[]);	matriz de ponteiros para tipo	int* count[];
tipo (*nome)[];	ponteiro para matriz do tipo	int (*count)[];
tipo& nome;	referência para o tipo	int& count;
tipo nome();	função que retorna o tipo	int count();
tipo*nome();	função que retorna ponteiro tipo*	int* count();
tipo*(nome());	função que retorna ponteiro tipo*	int*(count());
tipo (*nome)();	Ponteiro para função que retorna o tipo	int (*count)()

Capítulo 6

Tipos

Neste capítulo veremos o que é um tipo, quais os tipos pré-definidos de C++, os tipos definidos em bibliotecas externas como a STL e a definição de tipos do usuário.

6.1 Introdução ao conceito de tipos

Um tipo é uma abstração de algo.

Os tipos podem ser de três modos: os tipos pré-definidos da linguagem C++ (char, int, float, double), os tipos definidos pelo programador e os tipos definidos em bibliotecas externas (como a STL).

6.2 Uso de tipos pré-definidos de C++

Os tipos pré-definidos pela linguagem podem ser vistos como classes definidas pelo criador do C++ e que estão escondidas de você. Assim, pode-se imaginar que existem as classes: class char{}, class int{}, class float{}, class double{}. Desta forma, um número inteiro é um objeto, um número float é um objeto e assim por diante. Veja o exemplo.

Exemplo:

```
//Esta criando dois objetos inteiros, int é o tipo
//x,y são os nomes dos objetos
int x,y;
//armazena valores nestes objetos
x = 4; y = 5;
//Esta criando um outro objeto, do tipo int, com nome z e igualando a x
int z = x;
```

Em resumo: você precisa entender que existe uma classe que define os números inteiros e esta classe faz parte de uma hierarquia, a hierarquia dos tipos numéricos. A maneira como se declara, se define, se usa e se elimina um objeto do tipo inteiro é semelhante ao que ocorre para todos os outros tipos.

Os tipos numéricos pré-definidos da linguagem C++ estão listados na Tabela 6.1, bem como o intervalo de valores suportados (unsigned=sem sinal, short= curto, long= longo). Veja na Figura X uma representação dos diferentes tipos.

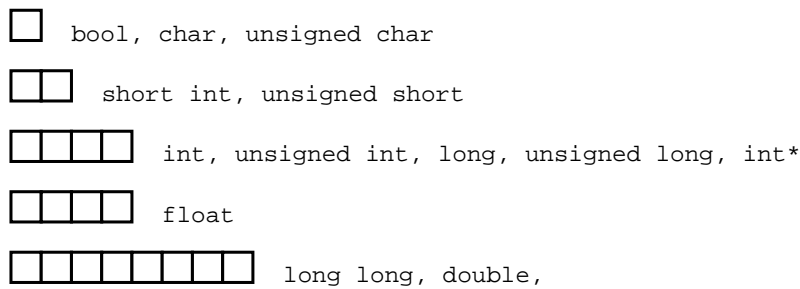


Figura 6.1: Tipos de dados e dimensões (sizeof).

Dê uma olhada no arquivo de biblioteca `<limits>`¹. Este arquivo contém variáveis que definem os limites para os tipos pré-definidos da linguagem.

Tabela 6.1: Tipos e intervalos.

Tipos Básicos	Características	bytes	Valor mínimo	valor máximo
bool	booleano	1	0	1
char	caracteres	1	-128	127
unsigned char	caracteres s/ sinal	1	0	255
short	inteiros	2	-32768	32767
unsigned short	int, peq. s/sinal	2	0	65535
int	inteiros	4	-2.147.483.648	+2.147.483.648
unsigned int	inteiro sem sinal	4	0	+4.294.295.000
long	inteiro grande	4	-2.147.483.648	+2.147.483.648
unsigned long	int,gde.c/sinal	4	0	+4.294.295.000
float	precisão simples, 7 dígitos	4	3.4e-38	3.4e+38
double	precisão dupla, 15 dígitos	8	1.7e-308	1.7e+308
long double	precisão dupla, 18 dígitos	10	3.4e-4932	3.4e+4932
enum	enumerados	2	-2.147.483.648	+2.147.483.648

A Tabela 6.2 mostra as diferenças entre as plataformas de 16 e 32 bits. Para float, double, long double os tamanhos não mudam.

Dica: O compilador g++ da gnu suporta o tipo long long, que representa um inteiro de 64 bits. O compilador da GNU é descrito no Capítulo 44.

Apresenta-se a seguir um programa que usa os tipos padrões de C++. Observe que este exemplo não tem nem entrada, nem saída de dados, não sendo necessária a inclusão (`#include`) de arquivos externos.

Listing 6.1: Tipos numéricos de C++.

¹Veja no manual do seu ambiente de desenvolvimento onde estão armazenados os arquivos da biblioteca de C++. No Linux estão em `/usr/include/g++`.

Tabela 6.2: Diferenças de tamanho dos objetos padrões de C++ nas plataformas de 16 e 32 bits.

	<i>16 bits</i>	<i>16 bits</i>	<i>32 bits</i>	<i>32 bits</i>
enum	-32768	+32768	-2.147.483.648	+2.147.483.648
unsigned int	0	65535	0	4.294.967.295
int	-32768	+32768	-2.147.483.648	+2.147.483.648

```

int main ()
{
    //Tipos padrões da linguagem
    //Tipo booleano
    //Intervalo 0 ou 1
    //1 bytes
    bool flag = 0;

    //Tipo char
    //Intervalo -128 -> +127
    //1 bytes
    char ch = 'b';

    //Tipo int
    //2 byts (16 bits), Intervalo (16bits)~ - 32768 -> + 32767
    //4 byts (32 bits), Intervalo (32bits)~ -2147483648 -> +2147483648
    int int_x = 777;

    //Tipo float
    //Intervalo +/- 3.4.e+/-38 (7 digitos precisão)
    //4 bytes
    float float_y = 3.212f;

    //Tipo double
    //Intervalo +/- 1.7e+/-308 (15 digitos precisão)
    //8 bytes
    double double_z = 12312.12312e5;

    //Tipo long double
    //Intervalo +/- 3.4e+/-4932 (18 digitos precisão)
    //10 bytes
    long double long_double_r = 1.2e-18;

    return 0;
}

/*
Novidade:
-----
Tipos: A linguagem C++ é altamente prototipada,
ou seja, dá uma importância muito grande ao tipo dos objetos.

Uso dos tipos padrões de C++
bool, char, int, float, double, long double

```

Criação de objetos numéricos e atribuição de valores a estes objetos.

Saída:

Este exemplo não inclui saída de dados.

**/*

Observe no exemplo a seguir o uso e as diferenças dos tipos int e unsigned int.

Listing 6.2: Diferenças no uso de inteiro com sinal (signed) e sem sinal (unsigned).

```
#include <iostream>

void main()
{
    {
        std::cout << "----->Testando uso de int" << std::endl;
        int x, y, z;
        std::cout << "Entre com int x (ex: 300):"; cin >> x;
        std::cout << "Entre com int y (ex: 500):"; cin >> y;
        cin.get();

        z = x + y;
        std::cout << "int z = x + y = " << z << std::endl;
        z = x - y;
        std::cout << "int z = x - y = " << z << std::endl;
    }

    std::cout << "----->Testando uso de unsigned int" << std::endl;
    unsigned int x,y,z;
    std::cout << "Entre com unsigned int x (ex: 300):"; cin >> x;
    std::cout << "Entre com unsigned int y (ex: 500):"; cin >> y; cin.get()
        ;
    z = x + y;
    std::cout << "unsigned int z = x + y = " << z << std::endl;
    z = x - y;
    std::cout << "unsigned int z = x - y = " << z << std::endl;

    //faz o teste abaixo, de forma a retornar o modulo da diferença
    //se x > y retorna z = x - y
    //se x <= y retorna z = y - x
    if( x > y)
        z = x - y ;
    else
        z = y - x ;

    //armazena a informação do sinal
    int sinal = x > y ? +1 : -1 ;

    //Cria objeto int para armazenar o resultado, observe que sinal é do
    tipo int
    int valor_z = sinal * z ;
    std::cout << "z = |x - y| = " << z << std:::
        endl;
    std::cout << "sinal de x - y = " << sinal << std:::
        endl;
}
```

```

        std::cout <<"int_valor_z= sinal*uz=" <<        valor_z <<        std::
            endl;
    }

    /*
    Novidade:
    -----
    -Uso de objeto do tipo int, com sinal (signed) e sem sinal (unsigned).

    -Uso do operador de controle if...else

    O operador if é usado para controlar a sequência de execução
    do programa. O if avalia uma expressão, se a mesma for verdadeira
    executa a linha abaixo, se a expressão for falsa, pula a linha abaixo.
        if( expresao)
            ação1 ;

    Fora o if, você também tem o if...else
        if(expresao)
            ação1 ;
        else
            ação2;

    No if...else, se a expressão for verdadeira executa ação 1,
    se for falsa executa ação 2.
    */

    /*
    Saída:
    -----
    [andre@mercurio lyx]$ ./a.out
    ----->Testando uso de int
    Entre com int x (ex: 300):300
    Entre com int y (ex: 500):500
    int z = x + y =800
    int z = x - y =-200
    ----->Testando uso de unsigned int
    Entre com unsigned int x (ex: 300):300
    Entre com unsigned int y (ex: 500):500
    unsigned int z = x + y =800
    unsigned int z = x - y =4294967096
    z=| x - y |=200
    sinal de x - y =-1
    int valor_z = sinal * z =-200
    */

    /*
    Análise da saída:

    Observe que a saída de
    z = x - y = 4294967096
    apresenta um valor esquisito: 4294967096
    isto ocorre porque z é um inteiro sem sinal
    que vai de 0-> 4294967295
    como x=300 y=500, x - y = -200
    */

```

*como z não pode armazenar valor negativo,
z fica com o valor 4294967096
que significa:*

```
4294967295 - 200 -1 (o zero) = 4294967096
*/
```

Dica: C++ tem um conjunto de operadores e de estruturas de controle. Como o conceito de operadores (+, -, *, ...) e de estruturas de controle já são bastante conhecidos², os mesmos foram incluídos na parte VI da apostila. Continue lendo a apostila normalmente. Se tiver problemas no entendimento de como funciona determinado operador ou estrutura de controle, aí sim, dê uma olhada no item específico na parte VI da apostila.

6.3 Uso de tipos do usuário

No capítulo classes, vamos definir os tipos do usuário. Os tipos do usuário são usados da mesma forma que os tipos padrões de C++.

Descreve-se abaixo, um exemplo preliminar de definição e uso de um tipo do usuário. O objetivo é mostrar que uma classe definida pelo usuário se comporta da mesma forma que os tipos pré-definidos da linguagem³.

Listing 6.3: Exemplo preliminar de definição de classe do usuário.

```
//Exemplo preliminar de definição de um tipo de usuário, através de uma classe:
//Definição do tipo do usuário Complexo, representa um número complexo
#include <iostream>

class Complexo
{
public:

//Construtor
    Complexo ():x (0), y (0)
    {
    };

//Sobrecarga operador+
    Complexo & operator+ (Complexo &);

//Método
    inline void Set (double _x, double _y)
    {
        x = _x;
        y = _x;
    }
};
```

²Todas as linguagens de programação incluem operadores e estruturas de controle.

³Uma pergunta que pode ser feita, é se as operações realizadas com os tipos pré-definidos (soma, subtração, multiplicação, ...), poderão também ser realizadas com os tipos do usuário?

A resposta é SIM.

Entretanto, você terá de sobrecarregar alguns operadores para que o seu tipo se comporte exatamente como um tipo pré-definido de C++. Para realizar uma soma com o objeto do usuário, sobrecarrega-se o operador+ (você vai aprender a implementar e usar a sobrecarga de operadores no Capítulo 18).

```

};

//Atributos
double x, y;           //Coordenadas x e y
};

//Exemplo de definição de um método da classe Complexo
//Exemplo que soma z=A+B, o número complexo A com B
Complexo & Complexo::operator+ (Complexo & p)
{
    Complexo* z = new Complexo;
    z->x = x + p.x;
    z->y = y + p.y;
    return (*z);
}

//Exemplo de criação e uso de um objeto do tipo Complexo
int main ()
{
    //Cria objetos a e b do tipo Complexo
    Complexo a, b;

    //Chama função Set do objeto a e b
    a.Set (5, 4);
    b.Set (2, 3);

    //cria um novo objeto complexo
    Complexo c;

    //soma dois complexos e armazena em c
    c = a + b;

    std::cout << "c(" << c.x << ", " << c.y << ")_=" << "\n"
    << "a(" << a.x << ", " << a.y << ")_+" << "\n"
    << "b(" << b.x << ", " << b.y << ")_" << std::endl;

    return 0;
}

/*
Novidade:
-Declaração de classe do usuário, a classe Complexo.
-Criação dos objetos Complexo a, b, c.
-Uso dos objetos criados c = a + b
*/
/*
Saída:
-----
[andre@mercurio Parte-II]$ ./a.out
c(7,7) = a(5,5) + b(2,2)
*/

```

Neste exemplo foi utilizado o conceito de sobrecarga de operador. Veja o conceito de operadores na seção C na página 553 e de sobrecarga de operadores na seção 18 na página 215.

Agora você já sabe usar os tipos padrões de C++ (char, int, float...), já tem idéia de como se

declara e se define um tipo do usuário. A seguir vamos ver um exemplo de uso de uma biblioteca externa, isto é, uso da biblioteca STL, a biblioteca standart de C++.

6.4 Uso de tipos definidos em bibliotecas externas (STL)

Uma biblioteca é um conjunto de objetos reunidos em um único arquivo. Você pode criar e usar suas próprias bibliotecas ou usar bibliotecas desenvolvidas por terceiros. Veremos como criar bibliotecas no Capítulo 46.

A standart template library (ou STL), é uma biblioteca avançada de C++. Todas as distribuições padrões de C++ incluem a STL.

No exemplo a seguir você verá o uso da classe vector, uma classe extremamente útil disponibilizada pela STL. Com a classe vector você elimina totalmente o uso de vetores e arrays no estilo de C.

No exemplo a seguir cria um vetor e solicita ao usuário a entrada de dados. Cada novo valor que o usuário entra é armazenado no vetor. Para encerrar a entrada de dados o usuário digita (ctrl + d, no Linux) e (ctrl + z, no Windows). A seguir o programa mostra os valores do vetor. O programa usa estruturas de controle, as mesmas são descritas no apêndice D.

Listing 6.4: Exemplo preliminar de uso da classe vector da biblioteca STL

```
//Classes para entrada e saída de dados
#include <iostream>

//Classe de vetores, do container vector
#include <vector>

//Definição da função main
int main ()
{
    //Cria vector, do tipo int, com nome v, um vetor de inteiros
    vector < int > v;

    int data;
    std::cout << "No_DOS_░░░░░_ctrl+z_encerra_░_entrada_de_dados." << std::endl;
    std::cout << "No_Mac_░░░░░_ctrl+d_encerra_░_entrada_de_dados." << std::endl;
    std::cout << "No_Linux_░_ctrl+d_encerra_░_entrada_de_dados." << std::endl;
    do
    {
        std::cout << "\nEntre_com_░_dado_(" << v.size () << "):";
        cin >> data;
        cin.get ();
        //acidiona ao final do vetor v o objeto data
        if (cin.good ())
            v.push_back (data);
    }
    while (cin.good ());

    //Acessa partes do vector usando funções front e back
    std::cout << "\nPrimeiro_░_elemento_░do_░vetor=_" << v.front ()
        << "\nÚltimo_░░░_elemento_░do_░vetor=_" << v.back () << std::endl;

    //Mostra o vetor
```

```

for (int i = 0; i < v.size (); i++)
{
    std::cout << "v[" << i << "]=" << v[i] << ' ';
}
std::cout << std::endl;

std::cout << (v.empty ()? "0_vetor_esta_vazio" : "0_vetor_não_esta_vazio") <<
std::endl;

//Chama função clear, que zera o vetor
v.clear ();
std::cout << (v.empty ()? "0_vetor_esta_vazio" : "0_vetor_não_esta_vazio") <<
std::endl;

std::cout << std::endl;
cin.get ();
return 0;
}

/*
Novidade:
Uso do container vector.
Uso dos métodos: push_back, size, empty, clear,
Uso dos operadores do..while e de for
Uso de cin.good() para verificar se a entrada foi correta.

0 Operador de controle do ...while();
-----
0 operador de controle do..while executa a sequência de comandos
dentro do bloco pelo menos 1 vez. A seguir verifica a expressão dentro do while
.

do
{
    comandos_a_executar
}
while (expressão);

enquanto a expressão for verdadeira executa a sequência
de comandos dentro do do{}while.
Observe a presença de ponto e vírgula após o while.

0 operador de controle for:
-----
Um comando for é utilizado para realizar um looping,
uma repetição de determinado comando diversas vezes.
0 protocolo de um comando for é da forma:

for (inicializacao; teste; incremento)
{}

Exemplo:

for ( int i = 0; i < 10; i++ )
    std::cout << " i = " i << std::endl;

```

No passo 1, inicializa a variável *i*, do tipo *int* com o valor 0.

No passo 2, verifica se *i* < 10.

No passo 3, se *i* < 10 executa a linha

```
std::cout << " i = " i << std::endl;
```

No passo 4, incrementa *i* (*i++*).

Daí em frente, repete os passos 2, 3, 4.

Quando a expressão (*i* < 10)

for falsa, encerra o for.

```
*/
```

```
/*
```

Saída:

```
-----
```

```
[andre@mercurio Parte-II]$ ./a.out
```

No DOS um ctrl+z encerra a entrada de dados.

No Mac um ctrl+d encerra a entrada de dados.

No Linux um ctrl+d encerra a entrada de dados.

Entre com o dado (0):1

Entre com o dado (1):2

Entre com o dado (2):3

Entre com o dado (3):

Primeiro elemento do vetor= 1

Último elemento do vetor= 3

v[0]=1 v[1]=2 v[2]=3

O vetor não está vazio

O vetor está vazio

```
*/
```

Bem, vimos um exemplo de uso dos objetos padrões de C++, como declarar, definir e usar um tipo do programador e como usar objetos de bibliotecas externas (como a STL). Dê uma revisada nos exemplos e veja que a forma como se declara, se define e se usa os três tipos de objetos é a mesma.

6.5 Vantagem da tipificação forte do C++

A tipificação forte obriga o programador a tomar um maior cuidado na declaração, definição e uso dos objetos, atributos e métodos. Em troca, tem uma garantia maior de que o código não apresenta problemas, pois com a tipificação forte, o compilador pode encontrar mais facilmente os erros no programa.

6.6 Sentenças para tipos

- Use `unsigned char` quando precisar trabalhar com números que vão de 0 a 255.
- Use `signed char` quando precisar trabalhar com números que vão de -128 a 127.
- Evite usar `unsigned`.

- Um tipo do usuário, ou um tipo de uma biblioteca externa, estão bem definidos, se puderem ser usados da mesma forma que um tipo padrão de C++.
- ³ O tipo long double possui 10 bytes de comprimento, é exatamente assim que o processador 80x87 trabalha com pontos flutuantes. Desta forma pode-se passar um objeto do tipo long double diretamente para programas em assembler.

Neste capítulo você aprendeu que a linguagem C++ é altamente prototipada. Aprendeu a criar e usar um objeto padrão de C++ (int, float). Aprendeu a criar objetos do usuário e a usá-los da mesma forma que os tipos padrões de C++. Também aprendeu a usar objetos de bibliotecas externas, como vector da STL.

Capítulo 7

Namespace

Neste capítulo apresenta-se o que é um namespace, como usar o espaço de nomes da biblioteca padrão de C++ (std), como definir e usar um namespace.

7.1 O que é um namespace ?

Como o próprio nome diz, significa espaço para nomes. Quando você monta seu programa utilizando bibliotecas externas podem ocorrer duplicações de nomes, isto é, um objeto definido em uma das bibliotecas tem o mesmo nome de um objeto definido por você.

Exemplo:

Você criou as funções `min()` e `max()`, que retornam o menor e maior valor de um vetor.

Mas a STL já tem estas funções.

Desta forma o compilador não sabe qual função `min()` você quer chamar.

Solucionar o problema da duplicação de nomes pode ser complexo, pois se estes nomes pertencerem a bibliotecas externas, você precisaria contactar os desenvolvedores destas bibliotecas para resolver os conflitos, ou renomear seus objetos e funções. O namespace veio para resolver este problema.

7.2 Usando o espaço de nomes da biblioteca padrão de C++ (std)

Para usar os objetos standard de C++ é preciso incluir a palavra `std` e a seguir o operador de resolução de escopo, isto é:

```
//Para usar uma função da std você usa
std::nomeFuncao();
//Para usar um objeto
std::nomeObjeto;
//Para chamar uma função da std com parâmetros
std::nomefuncao(std::nomeObjeto);
```

Nas listagens de código já apresentadas usamos:

```
int x = 3 ;
std::cout << " entre com x : ";
std::cin >> x ;
std::cin.get();
std::cout << " x = " << x << std::endl;
```

Pode-se utilizar os objetos standards de C++ diretamente, isto é, sem o uso de `std::`, para tal basta colocar a declaração `using namespace std` no início do programa.

Exemplo:

```
//Declara que vai usar os objetos standart de C++
using namespace std;
int x = 3 ;
cout << " entre com x : ";
cin >> x ;
cin.get();
cout << " x = " << x << endl;
```

7.3 Definindo um namespace²

Todo arquivo de código, deve ter uma declaração namespace indicando um nome geral para os códigos que estão sendo desenvolvidos. Veja o protótipo.

Protótipo:

```
namespace NomeNamespace
{
//Declarações de atributos e métodos
tipo nome;
retorno Nomefunção ( parâmetros );
}
//Definindo métodos de um namespace
void retorno NomeNamespaceNomefunção ( parâmetros )
{...};
```

Veja na listagem a seguir um exemplo de definição e uso de um namespace.

Listing 7.1: Definindo e usando um namespace.

```
//Exemplo: Definindo e usando um namespace
#include <iostream>

//Objeto x global
int x = 3;

//cria um bloco namespace com o nome teste
namespace teste
{
    const int x = 7;
    void Print ();
}
```

```

    namespace teste2
    {
        int y = 4;
    }
}

//função main
int main ()
{
    std::cout << x << std::endl;           //usa x global
    std::cout << teste::x << std::endl;    //usa x do bloco namespace
    std::cout << teste::teste2::y << std::endl;

    teste::Print();
    return 0;
}

//definição da função Print do namespace teste
void teste::Print ()
{
    std::cout << "\nfunção print do namespace" <<std:: endl;
    std::cout << x << std::endl;           //x do namespace
    std::cout <<::x << std::endl;         //x global
    std::cout << teste2::y << std::endl;
}

/*
Novidade:
Definição e uso de namespace
*/
/*
Saída:
-----
[root@mercurio Cap3-P00UsandoC++]# ./a.out
3
7
4

função print do namespace
7
3
4
*/

```

Observe que um espaço de nomes é um escopo. A definição do escopo e da visibilidade dos objetos em C++ será vista na seção 8.2 e no apêndice B.3.

7.4 Compondo namespace²

Você pode compor dois ou mais namespaces. Veja o exemplo.

```

Exemplo:
//Compondo namespaces
namespace lib1{...};

```

```
namespace lib2{...};
namespace minhaLib
{
using namespace lib1;
using namespace lib2;
...};
```

7.5 Sentenças para namespace

- Funções definidas dentro do namespace são visíveis entre si.
- Pode-se definir o uso de cada objeto do namespace

Exemplo:

```
/*A diretiva using sdt::cout;
define que seu programa vai usar o objeto std::cout
e o mesmo pode ser chamado diretamente, isto é
cout << 'x';*/
using sdt::cout;          //diretiva
cout << 'mensagem';
```

- Em arquivos de cabeçalho (*.h) nunca use **using namespace std;**. Nos arquivos *.h use `std::cout`. A linha **using namespace std;** deve ser usada apenas nos arquivos *.cpp.
- Pode-se criar um sinônimo para um namespace

```
namespace lib = Minha_lib_versao_x_y_z;
```

 Use um nome de namespace grande e depois crie um sinônimo para ele.
- Para acessar as variáveis e funções definidas em um namespace use o operador de resolução de escopo (::).
- Um namespace pode ser criado sem um nome

```
namespace {...}
```
- ²Observe que para acessar variáveis e funções definidas dentro de um namespace, usa-se o mesmo formato utilizado para acessar atributos de uma classe.
- ²Use namespace para grupos de classes que formam um assunto. Isto é, para reunir grupos de classes que representam um assunto em comum.

Capítulo 8

Classes

Vimos no capítulo 1 que a classe é a unidade de encapsulamento dos atributos e dos métodos. Dentro da classe é que declaramos os atributos, os métodos e seus controles de acesso. Neste capítulo vamos ver o protótipo para declarar e definir uma classe. Como usar as palavras chaves `public`, `protected` e `private` para encapsular os atributos e métodos da classe.

8.1 Protótipo para declarar e definir classes

Veja a seguir o protótipo geral para declaração de uma classe, de seus atributos e métodos.

Verifique que existem diferentes tipos de atributos e de métodos. Ao lado do formato um comentário informando a seção onde o mesmo vai ser apresentado.

Protótipo:

```
class TNome
{
//Atributos
tipo nome;           //atributos de objeto, seção 9.2
static tipo nome;   //atributos estáticos seção 9.3
const tipo nome;    //atributos const, seção 9.4
mutable tipo nome;  //atributos mutable, seção 9.5
volatile tipo nome; //atributos com volatile, seção 9.6
//Métodos
tipo função(parâmetros);           //métodos normais seção 10.4
tipo função(parâmetros) const ;    //métodos const, seção 10.5
static tipo função(parâmetros);    //métodos estáticos, seção 10.6
inline tipo função(parâmetros);    //métodos inline, seção 10.7
virtual tipo função(parâmetros);    //métodos virtuais, seção 16.2
virtual tipo função(parâmetros)=0;  //métodos virtuais puros, seção
};
```

Importante, observe a presença de `(;)` no final do bloco que declara a classe.

Exemplo:

```
//-----TEndereco.h
```

```
#include <string>
class TEndereco
{
//-----Atributo
int numero;
string rua;
//-----Métodos
int Getnumero();
string Getrua();
};
```

8.2 Encapsulamento em C++ usando o especificador de acesso

Para a análise orientada a objeto, encapsulamento é o ato de esconder do usuário informações que não são de seu interesse. O objeto é como uma caixa preta, que realiza determinada operação mas o usuário não sabe, e não precisa saber, exatamente como. Ou seja, o encapsulamento envolve a separação dos elementos visíveis de um objeto dos invisíveis.

A vantagem do encapsulamento surge quando ocorre a necessidade de se modificar um programa existente.

Para implementar o conceito de encapsulamento, C++ oferece as palavras chaves `public`, `protected` e `private`. Veja a seguir quando utilizar `public`, `protected` e `private` em suas classes.

public: significa que o atributo ou método faz parte da interface do objeto, podendo ser acessada a qualquer instante.

protected: significa protegida de acesso externo. Só pode ser acessada pelos métodos da classe e pelos métodos das classes derivadas.

private: só pode ser acessada pelos métodos da classe (métodos internos da classe e métodos `friend`¹).

Para aumentar o encapsulamento da classe, declare tudo como privado. Só deixe como público o que for essencial e fizer parte da interface da classe. Evite atributos protegidos, atributos protegidos podem ser usados em diversas classes derivadas, sendo difícil identificar o impacto de mudanças no mesmo. Não existem restrições para métodos protegidos.

O uso de `public`, `protected` e `private` será esclarecido através dos exemplos apresentados.

8.3 Classes aninhadas²

Podemos declarar classes aninhadas (classes dentro de classes).

```
Exemplo:
class A
{
```

¹Métodos `friend` serão descritos no capítulo 17.


```
int x;  
class XX {int y;}; //classe aninhada  
};
```

8.4 Sentenças para classes

- Para identificar as classes e seus relacionamentos, faça associações diretas com conceitos do mundo real.
- Uma classe definida pelo usuário é um tipo do usuário.
- Quando você cria uma classe esta definindo um novo tipo. Você deve incluir na documentação do programa o conceito e a forma de uso do novo tipo.
- Todos os membros de uma classe tem de ser declarados em seu interior.
- Um mesmo nome não pode ser dado a um método e a um atributo.
- Uma classe A qualquer, não pode conter um objeto do tipo A, mas somente um ponteiro para A. Antes do final da declaração de uma classe seu nome só pode ser utilizado se o tamanho da classe não for necessário.
- Crie classes pequenas para realizar tarefas pequenas.
- Ao criar suas classes dê a elas um formato simples. Uma classe TSolver deve representar um solver.
- Em problemas de engenharia, associe classes a conceitos reais.

Exemplo:

uma classe edificio, janela, porta,..., e assim por diante.

- Uma classe é um conjunto de atributos (variáveis ou objetos) reunidos com um conjunto de métodos (funções). Os atributos que incluo na classe são aqueles que fazem sentido a classe. O mesmo para os métodos.
- A classe não é um objeto, é uma descrição do objeto (definição da forma e conteúdo do objeto).
- Os atributos que compõem a classe podem ser de diferentes tipos. No exemplo usa duas strings e um int.
- ² Uma classe com atributos const ou referências tem de ter obrigatoriamente um construtor para inicializar estes atributos.
- ² Se um método tem inicializadores, isto é, atributos com valores pré-definidos. Os inicializadores devem ficar visíveis na classe (arquivo *.h), pois quem usa a classe olha o arquivo de cabeçalho (*.h), e nem sempre tem acesso ao arquivo de implementação (*.cpp).

- ² Quando você escreve `TNomeClasse`, você está definindo a classe, entretanto, por razões históricas, costuma-se chamar de declaração da classe. O ideal seria, "estou declarando um conjunto de instruções (atributos e métodos) que definem um novo tipo do usuário, ou ainda, é uma declaração porque não reserva espaço de memória, é uma definição porque define um novo tipo.
- ² Observe que a chamada `f().g()` é legal se `f()` retornar um objeto pela qual `g()` pode ser chamada.

Capítulo 9

Atributos

Neste capítulo apresenta-se o conceito de atributos, o protótipo para declarar e definir atributos. O que são atributos de objeto, de classe, como modificar os atributos de objeto e de classe com as palavras `const`, `mutable` e `volátil`.

9.1 Protótipo para declarar e definir atributos

Os atributos são definidos dentro da classe. Um atributo pode ser um tipo pré-definido da linguagem, um tipo definido pelo programador ou um tipo de uma biblioteca externa como a STL.

Apresenta-se a seguir o protótipo para declaração de um atributo normal, de um atributo de classe (ou estático) e de atributos `const`, `mutable` e `volátil`.

Protótipo:

```
class TNome
{
//Atributos
tipo nome;           //atributos de objeto, seção 9.2
static tipo nomeA;   //atributos de classe estáticos, seção 9.3
const tipo nome;     //atributos const, seção 9.4
mutable tipo nome;   //atributos com mutable, seção 9.5
volatile tipo nome;  //atributos com volatile, seção 9.6
};
tipo TNome::nomeA;   //definição de atributo estático
```

9.2 Atributos de objeto

Um atributo de objeto é um atributo declarado dentro da classe sem o uso do modificador de tipo `static`.

Para criar um atributo de objeto coloca-se o tipo seguido do nome do atributo.

Exemplo:
`int x;`

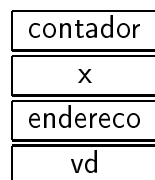
O exemplo a seguir ilustra a declaração dentro da classe de alguns atributos.

Exemplo:

```
#include <iostream>
#include <string>
#include <vector>
class B
{
public:
//int é um tipo padrão de C++
int contador;
//float é um tipo padrão de C++
float x;
//O tipo TEndereco foi definido no exemplo anterior é um tipo do programador
TEndereco endereco;
//string é um tipo definido em <string>
string nome;
//O tipo vector é um tipo da biblioteca padrão stl
vector < double >  vd;
};
//Dentro de main cria e usa um objeto do tipo B
void main()
{
B b;
b.x = 3.1;
b.nome = "joao";
b.vd[0] = 34.5;
cout << "b.x=" << b.x << " b.nome=" << b.nome;
cout << " b.vd[0]=" << b.vd[0];
}
```

Observe que usamos tipos padrões de C++ (int e float), tipos do usuário (TEndereco) e tipos da STL (vector). Observe na Figura 9.1 como fica um objeto do tipo B na memória.

Figura 9.1: Como fica o objeto b na memória.



9.3 Atributos de classe (estáticos)

Existem dois tipos de atributos dentro de uma classe, os atributos de objeto (individual, é armazenado no objeto) e os atributos de classe (coletivos, é armazenado na classe). O objetivo dos atributos de classe é possibilitar o compartilhamento do mesmo por todos os objetos criados.

Para criar um atributo de classe coloca-se a palavra chave `static` seguida do tipo e do nome do atributo.

Exemplo:

```
static int contador;
```

O exemplo a seguir ilustra a declaração e a definição de atributos de classe (estáticos).

Exemplo:

```
#include "TEndereco.h"
class TUniversidade
{
public:
//Atributo de objeto, do tipo int, com nome numeroAlunos
int numeroAlunos;
//Atributo de objeto, do tipo Endereco, com nome endereco
Endereco endereco;
//Atributo de classe (static), do tipo string, com nome pais
static string pais ;
};
string TUniversidade::pais = "Brasil";
void main()
{
//Cria dois objetos do tipo TUniversidade
TUniversidade ufsc,unicamp;
}
```

Observe que atributos estáticos precisam ser definidos fora da classe. Observe a forma da definição.

```
//tipo nomeClasse::nomeAtributo = valor;
string TUniversidade::pais = "Brasil";
```

Veja na Figura 9.2 como fica a memória para os objetos `ufsc` e `unicamp`. Observe que tanto o objeto `ufsc` quanto o objeto `unicamp` tem os atributos `numeroAlunos` e `endereco`, mas o atributo `pais` é armazenado na classe e compartilhado pelos dois objetos.

Figura 9.2: Como fica o objeto na memória quando a classe tem atributos estáticos.

ufsc	unicamp	Classe TUniversidade
		pais
numeroAlunos	numeroAlunos	
endereco	endereco	

9.3.1 Sentenças para atributos de classe

- O objetivo dos membros estáticos em classes é eliminar o uso de variáveis globais.
- Membros estáticos podem ser usados para compartilhar atributos entre objetos da mesma classe.
- Atributos estáticos e públicos podem ser acessados sem a necessidade de se criar um objeto da classe.

Exemplo:

```
string pais = TUniversidade::pais;
```

- Se o atributo for estático e publico, o mesmo pode ser acessado externamente. Basta passar o nome da classe, o operador de resolução de escopo (::) e o nome do atributo.
- ²Um objeto estático definido dentro de um método é criado na primeira execução do método e destruído no final do programa.
- ³ Uma classe local não pode conter membros estáticos.

9.4 Atributos const

O objetivo de um atributo const é fornecer ao objeto um atributo que ele vai poder acessar, mas não vai poder alterar, ou seja, é usado para criação de objetos constantes. Um atributo const pode ser inicializado nos construtores¹ da classe não podendo mais ser alterado.

Exemplo:

```
class TMath
{
//Atributo estático e constante, do tipo float, com nome pi
//pertence a classe a não pode ser mudado
static const float pi ;

//Atributo nome e constante, do tipo int, com nome max
//pertence ao objeto, não pode ser mudado
const int max;

//Inicializa atributo max no construtor
X(int par_max):max(par_max){};
};
const float TMath::pi = 3.141516;
```

No exemplo acima cria um atributo de classe e constante. PI pertence a classe (static) e não pode ser modificado. O mesmo é definido e inicializado na linha:

```
const float TMath::pi = 3.141516;
```

O atributo max pertence ao objeto, é inicializado no construtor da classe e não pode ser alterado (é constante).

¹Os construtores serão descritos na seção 13.2.

9.5 Atributos com mutable²

A palavra chave mutable pode ser utilizada em classes como uma alternativa ao `const_cast`. Se uma classe tem um método membro `const` o mesmo não pode alterar os atributos da classe. Uma forma de contornar isto é definir o atributo como mutable. Veja o exemplo:

```
Exemplo:
class Teste
{
mutable int x;
void Altera_x()const;
}
//Abaixo, embora a função seja const,
//x é alterado pois foi declarado como mutable.
void Altera_x()const {x++;};
```

A utilidade real de mutable ocorre no caso em que a função usa diversos atributos do objeto, mas você quer ter certeza de que os mesmos não sejam alterados e por isso declara a função como `const`. Mas por algum motivo, um único atributo precisa ser alterado, bastando para tal defini-lo como mutable.

Dica: mutable significa pode mudar, mesmo sendo declarado como `const`.

9.6 Atributos com volatile³

A palavra chave volátil é usada para dar a um atributo o status de volátil. Um atributo volátil pode mudar de forma inesperada.

9.7 Inicialização dos atributos da classe nos construtores²

Os atributos do objeto podem ser inicializados no construtor. O formato é dado por:

Protótipo:

```
NomeClasse(parametros) : atributo_1(valor), atributo_2(valor),...,atributo_n(valor)
{definição do construtor};
```

Veja a seguir um exemplo de inicialização dos atributos da classe:

```
Exemplo:
class CNome
{
int a, b, c;
//Depois do nome do construtor e de seus parâmetros o uso de
//: e a seguir os atributos com seus valores separados por vírgula.
CNome(): a(0),b(0),c(0)
{};
};
```

Observe que ao inicializar o atributo no construtor o mesmo é inicializado como uma cópia do valor passado, o que é mais rápido do que atribuir um valor ao atributo dentro do construtor.

Exemplo:

```
NomeClasse(int _a, int _b): a(_a), b(_b){} ; // + rápido
NomeClasse(int _a, int _b){ a= _a; b = _b;} ; //+ lento
```

9.8 Sentenças para atributos

- Observe que você pode combinar algumas especificações, isto é,

Exmplo:

```
int x;
const int y;
mutable static const int x;
```

- Se o atributo for const, o mesmo não muda.
- ²Se o atributo for o mesmo para todas as classes da hierarquia, use static.

Capítulo 10

Métodos

Neste capítulo apresenta-se como declarar, definir e usar métodos em uma classe. Como funciona a passagem de parâmetros por cópia, por referência e por ponteiro. O uso de argumentos pré-definidos e os métodos normais, constantes e estáticos.

10.1 Protótipo para declarar e definir métodos

Veja a seguir o protótipo para declarar e definir os métodos em uma classe.

Protótipo:

```
//-----TNome.h
class TNome
{
//Métodos
tipo função(parâmetros); //métodos normais, seção 10.4
tipo função(parâmetros) const; //métodos const, seção 10.5
static tipo função(parâmetros); //métodos estáticos, seção 10.6
inline tipo função(parâmetros); //métodos inline, seção 10.7
virtual tipo função(parâmetros); //métodos virtuais, seção 16.2
virtual tipo função(parâmetros)=0; //métodos virtuais puros, seção
};
//-----TNome.cpp
//Definição de um método da classe
tipo NomeClasse::Função(parâmetros)
{
//implementação do método ...
return(tipo);
}
```

Observe que os métodos de uma classe são declarados dentro da classe (nos arquivos *.h), e definidos fora da classe (nos arquivos *.cpp)¹.

¹com excessão dos métodos inline explicitos.

A declaração dos métodos normais, const, estáticos, inline e virtuais é diferente. Mas a definição é igual. O acesso aos métodos da classe pode ser modificado com as palavras chaves public, protect e private.

10.2 Declaração, definição e retorno de um métodos

Um método recebe como parâmetros de entrada um conjunto de objetos, realiza determinada seqüência de operações e a seguir retorna um objeto.

As tarefas realizadas por um método são:

- Receber uma lista de objetos (parâmetros).
- Executar um conjunto de tarefas.
- Retornar apenas um objeto.

Exemplo:

```
//declaração e definição da classe
class C
{
//protótipo do método soma
int soma(a,b);
};
```

Descreve-se a seguir cada uma destas tarefas.

10.2.1 Declaração de um método

Um protótipo de um método é a declaração de seu retorno, seu nome e seus parâmetros, antes de sua definição.

- Os protótipos dos métodos são declarados dentro das classes (no arquivo de cabeçalho, *.h).
- A vantagem dos protótipos é que eles facilitam o trabalho do compilador, ou seja, auxiliam na identificação da chamada de métodos com parâmetros errados.
- Na declaração de um método o nome dos argumentos é opcional.

10.2.2 Definição de um método

A definição de um método é a implementação de seu código.

Exemplo:

```
//definição do método soma da classe C
int C::soma(a,b)
{
return a+b;
}
```

```
int main()
{
    //Cria objeto do tipo int com nome x
    int x = 3;
    int y = 4;
    //Cria objeto do tipo C com nome obj
    C obj;
    //uso do método soma de obj
    int z = obj.soma(x,y);
    cout << " soma = " << z << endl ;
    return soma;
}
```

10.2.3 Retorno de um método

Todo método deve ter um retorno. Quando você não quer nenhum tipo de retorno, deve especificar o tipo void.

```
Exemplo:
class CC
{
    //Método com retorno do tipo int, com nome f e com parametro int x
    int f(int x);
    //Método com retorno do tipo double
    double sqrt(double x);
    //Método sem retorno
    void funcao();    //C++
};
```

O uso de void como retorno significa que o método não tem retorno.

10.3 Passagem dos parâmetros por cópia, por referência e por ponteiro

Os parâmetros de um método podem ser passados por cópia, referência ou ponteiro.

Por cópia: é passada uma cópia do objeto é a condição default em métodos.

É mais lento porque precisa criar uma cópia de cada objeto passado como parâmetro. Como é criada uma cópia, o objeto original não sofre nenhuma alteração.

Declaração	Chamada	Definição
int f1 (int a);	int b=1; int c=f1(b)	int NomeClasse::f1(int a){return ++a;}

Por referência: é passada uma referência do objeto.

É mais rápido, visto que tem acesso direto aos parâmetros (sem criar uma cópia dos mesmos). Observe que o objeto passado pode sofrer alterações dentro do método.

Declaração	Chamada	Definição
void f3 (float& c);	float c; f3(c);	void NomeClasse::f3(float &c){c=5;}

Por ponteiro²: é passado um ponteiro² para o objeto. Declara-se como parâmetro um ponteiro, que é utilizado dentro do método para acessar o objeto. O objeto passado através do ponteiro pode sofrer alterações.

Declaração d	Chamada	Definição
void f2 (float *b);	float b=1; f2(&b);	void NomeClasse::f2(float*b){*b=5;}

Veja a seguir um exemplo ilustrando a passagem de parâmetros. Não se preocupe se não entender a parte que usa ponteiros, depois de ler o capítulo de ponteiros, releia este exemplo.

Listing 10.1: Passando parâmetros por valor, referência e ponteiro.

```
//-----*.h
#include <iostream>
using std::cout;
using std::endl;

class Teste
{
public:

//Declaração de um método que recebe parâmetros por valor (int)
    int Soma_Valor (int, int);

//Declaração de um método que recebe parâmetros por referência (int&)
    int Soma_Referencia (int &, int &);

//Declaração de um método por ponteiro (int*)
    int Soma_Ponteiro (int *, int *);

//Declaração de um método que recebe parâmetros com referencia para um ponteiro
    (int * &)
    int Soma_ReferenciaPonteiro (int *&, int *&);
};

//-----*.cpp
//Definição de um método por valor (int)
//Os objetos x e y serão uma cópia dos objetos passados.
//Alterar x e y dentro do método Soma_Valor não altera os objetos passados
//vantagem: não altera o objeto passado,
//desvantagem: não altera o objeto passado (depende do que você quer).
int Teste::Soma_Valor (int x, int y)
{
    int soma = x + y;
    x = 5; //inútil, usado apenas para mostrar que x,y
    y = 7; //externos não são alterados
    return soma;
}
```

²Um ponteiro é um objeto que aponta para outro objeto, podendo ser usado para alterar os valores armazenados no objeto apontado. Veja descrição dos ponteiros no capítulo 12.

```

//Definição de um método com referência (int &)
//Com referência é passado o próprio objeto
//de forma que modificar x e y dentro do método Soma_Referencia,
//modifica o que foi passado.Use quando quiser alterar o objeto passado.
int Teste::Soma_Referencia (int &x, int &y)
{
    int soma = x + y;
    x = 55;
    y = 77;
    return soma;
}

//Definição de um método por ponteiro (int*)
//Note que precisa usar *x e *y para acessar o conteúdo dos objetos passados.
//Os ponteiros *x e *y são criados na chamada do método
//observe que altera os valores externos de x e y
int Teste::Soma_Ponteiro (int *x, int *y)
{
    int soma = *x + *y;
    *x = 555;
    *y = 777;
    return soma;
}

//Definição de um método referência para um ponteiro (int*&)
//Use quando já tiver ponteiros e desejar passar os próprios
//ponteiros e não uma cópia deles
//observe que altera os valores externos de x e y
int Teste::Soma_ReferenciaPonteiro (int *&x, int *&y)
{
    int soma = *x + *y;
    *x = 5555;
    *y = 7777;
    return soma;
}

int main ()
{
    int a = 1;
    int b = 2;
    cout << "a=" << a << " b=" << b << endl;

    Teste obj;

    cout << "Soma=" << obj.Soma_Valor (a, b) << endl;
    cout << "Após chamar Soma_Valor(a,b); a=" << a << " b=" << b << endl;

    cout << "Soma=" << obj.Soma_Referencia (a, b) << endl;
    cout << "Após chamar Soma_Referencia(a,b); a=" << a << " b=" << b << endl;

    cout << "Soma=" << obj.Soma_Ponteiro (&a, &b) << endl;
    cout << "Após chamar Soma_Ponteiro(&a,&b); a=" << a << " b=" << b << endl;

    int *pa = &a;
    int *pb = &b;

```

```

cout << "Soma=" << obj.Soma_ReferenciaPonteiro (pa, pb) << endl;
cout << "Após chamar Soma_ReferenciaPonteiro(pa,pb); a=" << a << " b=" << b
    << endl;
return 0;
}

/*
Novidade:
-----
-Aborda a passagem de parâmetros para métodos
por valor, por referência, por ponteiro e usando referência para o ponteiro.
*/
/*
Saída:
[andre@mercurio Cap2-Sintaxe]$ ./a.out
a=1 b=2
Soma=3
Após chamar Valor(a,b); a=1 b=2
Soma=3
Após chamar Referencia(a,b); a=55 b=77
Soma=132
Após chamar Ponteiro(&a,&b); a=555 b=777
Soma=1332
Após chamar ReferenciaPonteiro(pa,pb); a=5555 b=7777
*/

```

10.3.1 Uso de argumentos pré-definidos (inicializadores)

O uso de argumentos pré-definidos consiste em atribuir valores iniciais aos parâmetros de um método. Assim, quando o método é chamado sem argumentos, serão usados os argumentos pré-definidos.

No exemplo abaixo o método `f` tem os parâmetros `a`, `b` e `c` previamente inicializados com os valores 4, 7 e 9.3, respectivamente. Observe que o método `f` pode ser chamado de diferentes formas e que o objeto que deixa de ser fornecido é aquele que está mais à direita.

Exemplo:

```

int NomeClasse::f(int a=4, int b=7, float c=9.3)
{
    return a + b + c;
}

//O método pode ser chamado das seguintes formas:
NomeClasse obj;
obj.f (77, 5, 66.6); //a=77, b=5, c=66.6
obj.f(33, 75);      //a=33, b=75,c=9.3
obj.f(67);          //a=67, b=7, c=9.3
obj.f();            //a=4, b=7, c=9.3

```

10.3.2 Sentenças para declaração, definição e retorno de métodos

- Evite usar parâmetros de métodos com nome igual ao de outros objetos, o objetivo é evitar ambiguidades.

- Em C++ todas os métodos precisam de protótipos. O protótipo auxilia o compilador a encontrar erros.
- Analise os parâmetros dos métodos. Se os mesmos não são alterados, devem ser declarados como `const`.
- Objetos grandes devem ser passados por referência ou por ponteiros.
- Veja a função `main()` e a entrada na linha de comando na seção E.2, funções recursivas na seção E.3.
- O uso de `void` como argumento significa que o método não tem argumento. O exemplo esclarece.

Exemplo:

```
//sem retorno e sem parâmetro
void funcao(void); //C
void funcao();    //C++
```

- Segurança: Se você quer ter certeza de que o parâmetro não vai ser alterado, deve passá-lo como referência constante, veja o exemplo abaixo.

Exemplo:

```
//0 especificador const informa que o objeto
//é constante e não pode ser alterado dentro do método
//Deste modo o método pode acessar o objeto/mas não pode modificá-la.
funcao(const tipo& obj);
```

- ²O retorno de um método pode ser uma chamada a outro método ou a um objeto.
- ²Performance: Parâmetros passados por referência aumentam a eficiência pois os valores não são copiados.
- ²Um objeto é criado por cópia quando:
 - 1-é um parâmetro de um método.
 - 2-é um retorno de um método.
 - 3-é lançado como uma exceção.

10.4 Métodos normais

Os métodos normais são declarados dentro da classe sem nenhum especificador adicional. Sem uso dos especificadores `inline`, `static`, `virtual` ou `const`.

Veja na Figura 10.1 o diagrama UML da classe `TPessoa`. A classe `TPessoa` é implementada na listagem a seguir, a mesma apresenta a declaração e uso de classes, atributos e métodos normais.

Listing 10.2: Classe com atributo e método normal.

```
//-----Arquivo TALuno.h
//-----Bibliotecas C/C++
#include <iostream>
```

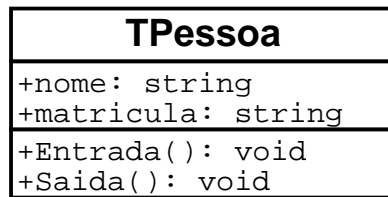


Figura 10.1: A classe TPessoa.

```

#include <string>
#include <vector>
using namespace std;

//-----Classe
/*
A classe TPessoa representa uma pessoa (um aluno ou um professor)
de uma universidade.
Tem um nome, uma matricula.
E métodos básicos para entrada e saída de dados.
*/
class TPessoa
{
//-----Atributos
public:
    //atributos normais
    string nome;
    string matricula;

//-----Métodos
public:
    //Uma função do objeto, altera as propriedades do objeto
    //Leitura dos atributos (nome, matricula)
    void Entrada();

    //Saida dos atributos (nome, matricula)
    void Saida() const;
};

//-----Arquivo TAluno.cpp
//Definição dos métodos
void TPessoa::Entrada()
{
    cout << "Entre com o nome do aluno: ";
    getline(cin,nome);

    cout << "Entre com a matricula do aluno: ";
    getline(cin,matricula);
}

void TPessoa::Saida() const
{
    cout << "Nome do aluno: " << nome << endl;
}

```



```

        cout << "Matricula_:_:" << matricula << endl;
    }

//-----Arquivo main.cpp
int main ()
{
    string linha="-----\n";

    const int numeroAlunos=5;

    //Cria um objeto do tipo TPessoa com nome professor
    TPessoa professor;

    cout << "Entre_com_o_nome_do_professor:_:";
    getline(cin,professor.nome);

    cout << "Entre_com_a_matricula_do_professor:_:";
    getline(cin,professor.matricula);

    //Cria um vetor de objetos do tipo TPessoa com nome aluno
    //com numero de elementos dados por numeroAlunos
    vector< TPessoa > aluno(numeroAlunos);

    for(int contador=0; contador < numeroAlunos; contador++)
    {
        cout << "Aluno_"<< contador <<endl;
        aluno[contador].Entrada();
    }

    cout << linha;
    cout << "RELAÇÃO_DE_PROFESORES_E_ALUNOS:_:"<<endl;
    cout << linha;

    cout << "Nome_do_professor:_:" << professor.nome << "\n";
    cout << "Matricula_:_:" << professor.matricula << "\n";

    for(int contador=0; contador < numeroAlunos; contador++)
    {
        cout << linha;
        cout << "Aluno_" << contador <<endl;
        aluno[contador].Saida();
    }
    cout << linha;

    cin.get();
    return 0;
}

/*Novidades:
Uso de tipos padrões de C++, de tipos do usuário e de tipos da STL

Uso de strings de C++:
-----

```

Neste exemplo utiliza-se a classe `string` de C++, definida em `<string>`. É uma classe padrão de C++, utilizada para manipular conjuntos de caracteres (mensagens).

Para usar uma `string` você deve incluir o cabeçalho `#include <string>`

Observe que a declaração de uma `string` é simples.

```
string nome_string;
```

Para armazenar algo na `string` faça:

```
nome_string = "conteúdo da string";
```

Para ler uma `string` do teclado :

```
cin >> nome_string;
```

Para ler do teclado toda uma linha e armazenar na `string`, use `getline`:

```
getline(cin, nome_string);
```

Este é seu primeiro programa orientado a objeto.

Simples ?

Você declarou uma classe.

```
class TPessoa...
```

Definiu as funções da classe

```
void TPessoa::Entrada()...
```

Criou objetos da sua classe

```
TPessoa professor;...
```

Usou o objeto

```
cout << "Nome do professor: " << professor.nome << "\n";
```

```
*/
```

10.5 Métodos const

Se um método da classe não altera o estado do objeto, não altera os atributos do objeto, ele deve ser declarado como `const`.

A declaração `const` instrue o compilador de que o método não pode alterar o estado do objeto, isto é, não pode alterar os atributos do objeto. Observe no exemplo a seguir, que a palavra chave `const` é colocada antes do `(;)` ponto e vírgula.

Exemplo:

```
class A
{
int valor;
//declaração
int GetValor() const;
};
//definição
int A::GetValor() const
{
return valor;
}
```

Apresenta-se a seguir um exemplo de uso de atributos e métodos const.

Listing 10.3: Classe com atributo e método const.

```
//-----Arquivo *.h
//-----Classe
class TNumeroRandomico
{
//-----Atributos
private:

    //Atributo normal
    double random;

    //Atributo constante
    const int semente_const;

//-----Métodos
public:
    //Construtor com parâmetros
    TNumeroRandomico (const int _semente_const = 1);

    //Método const não muda o estado do objeto
    double GetRandomNumber () const
    {
        return random;
    };
    const int Getsemente () const
    {
        return semente_const;
    };

    //Atualiza o número randômico
    void Update ();

};

//-----Arquivo *.cpp
#include <iostream>
using std::cout;
using std::endl;

#include <iomanip>
using std::setw;

#include <cstdlib>

//Construtor
TNumeroRandomico::TNumeroRandomico (const int _semente_const = 1):
semente_const (_semente_const)
// <-Precisa inicializar o atributo constante
{
    random = 0;
    srand (semente_const);
};
```

```

}

//Update gera um novo número randômico e armazena em random
void TNumeroRandomico::Update ()
{
    random = rand ();
}

//Função main
int main ()
{
    cout << "\nEntre com uma semente:" << endl;
    int semente = 0;
    cin >> semente;
    cin.get ();
    TNumeroRandomico gerador (semente);

    cout << "Valor da semente:" << gerador.Getsemente () << endl;
    cout << "Valor inicial:" << gerador.GetRandomNumber () << endl;

    for (int a = 0; a < 15; a++)
    {
        gerador.Update ();
        cout << "gerador.GetRandomNumber(" << setw (3) << a << ")=" << setw (15)
            << gerador.GetRandomNumber () << endl;
    }

    return 0;
}

/*
Novidade:
-----
- Mostra o uso na classe de atributo e método constante.

- Mostra o uso do método construtor para inicializar atributos do objeto.
TNumeroRandomico::TNumeroRandomico( const int _semente_const = 1 )
    :semente_const(_semente_const)

- Neste exemplo, duas funções da biblioteca de C,
a srand(semente_const); e a rand();
e dois atributos, semente_const e random, são encapsulados
(agrupados) para formar um objeto RandomGenerator, um objeto gerador de números
aleatórios.
*/

```

10.6 Métodos estáticos

Vimos que existem dois tipos de atributos, os atributos de classe e de objeto. Se você montar um método que só opera sobre os atributos estáticos da classe, pode declará-lo como sendo um método estático.

Um método estático e público, pode ser acessado sem um objeto da classe, basta colocar o nome da classe o operador de resolução de escopo (::) e o nome do método.

Exemplo:

```
//Abaixo o método estático é acessado sem um objeto da classe.
tipo x = NomeClasse::NomeMetodoEstatico();
```

Você pode passar informações de um objeto para outro através de atributos e métodos estáticos da classe.

Veja a seguir um exemplo com atributos e métodos estáticos.

Listing 10.4: Classe com atributo e método estático.

```
//-----Arquivo TAluno.h
//-----Bibliotecas C/C++
#include <iostream>
#include <string>
#include <vector>
using namespace std;

//-----Classe
/*
A classe TPessoa representa uma pessoa (um aluno ou um professor)
de uma universidade.
Tem um nome, uma matricula e um IAA.
E métodos básicos para entrada e saída de dados.
*/
class TPessoa
{
//-----Atributos
public:
    string nome;
    string matricula;
    float iaa;

private:
    static int numeroAlunos;

public:

//-----Métodos
    //Método do objeto, altera as propriedades do objeto
    //Leitura dos atributos (nome, matricula)
    void Entrada ();

    //Saida dos atributos (nome, matricula, iaa)
    void Saida () const;

    //Um método estático só pode alterar atributos estáticos
    static int GetnumeroAlunos ()
    {
        return numeroAlunos;
    }
};

/*
//Atributo estático é aquele que pertence a classe e não ao objeto
//e precisa ser definido depois da classe
```

```

*/
int TPessoa::numeroAlunos = 0;

//-----Arquivo TAluno.cpp
//Definição dos métodos
void TPessoa::Entrada ()
{
    cout << "Entre com o nome do aluno: ";
    getline (cin, nome);

    cout << "Entre com a matricula do aluno: ";
    getline (cin, matricula);

    cout << "Entre com o IAA do aluno: ";
    cin >> iaa;
    cin.get ();
}

void TPessoa::Saida () const
{
    cout << "Nome do aluno: " << nome << endl;
    cout << "Matricula: " << matricula << endl;
    cout << "iaa: " << iaa << endl;
}

//-----Arquivo main.cpp
int main ()
{
    string linha =
        "-----\n";

    cout << "Entre com o número de alunos da disciplina (ex=3): ";
    int numeroAlunos;
    cin >> numeroAlunos;
    cin.get ();

    //Cria um objeto do tipo TPessoa com nome professor
    TPessoa professor;

    //Cria um vetor de objetos alunos do tipo TPessoa
    vector < TPessoa > aluno (numeroAlunos);

    cout << "Entre com o nome do professor: ";
    getline (cin, professor.nome);

    cout << "Entre com a matricula do professor: ";
    getline (cin, professor.matricula);

    for (int contador = 0; contador < aluno.size (); contador++)
    {
        cout << "Aluno " << contador << endl;
        aluno[contador].Entrada ();
    }
}

```

```

cout << linha;
cout << "RELAÇÃO DE PROFESSORES E ALUNOS:" << endl;
cout << linha;

cout << "Nome do professor:" << professor.nome << "\n";
cout << "Matricula:" << professor.matricula << "\n";

for (int contador = 0; contador < aluno.size (); contador++)
{
    cout << linha;
    cout << "Aluno" << contador << endl;
    aluno[contador].Saida ();
}

cin.get ();
return 0;
}

/*
Revisão:
Os atributos que compoem a classe podem ser de diferentes tipos.
No exemplo usa duas strings, um float e um static int.
Os métodos são um contrutor e um destrutor (vistos posteriormente)
uma função de Entrada e outra de Saída.
*/
/*
Novidade:
-----
Uso de atributos e métodos estáticos
*/

/*
Saída:
-----
[andre@mercurio Cap3-P00UsandoC++]$ ./a.out
Entre com o número de alunos da disciplina (ex =3):2
Entre com o nome do professor: J.A.Bellini
Entre com a matricula do professor: 1
Aluno 0
Entre com o nome do aluno: Joao da Silva
Entre com a matricula do aluno: 2
Entre com o IAA do aluno: 123
Aluno 1
Entre com o nome do aluno: Pedro
Entre com a matricula do aluno: 2
Entre com o IAA do aluno: 32
-----
RELAÇÃO DE PROFESSORES E ALUNOS:
-----
Nome do professor: J.A.Bellini
Matricula : 1
-----

```

```

Aluno 0
Nome do aluno: Joao da Silva
Matricula : 2
iaa : 123

```

```

-----
Aluno 1
Nome do aluno: Pedro
Matricula : 2
iaa : 32
*/

```

10.7 Métodos inline

Quando você define um método, o compilador reserva um espaço de memória para o mesmo. Este espaço é ocupado pelo código do método. Quando você chama um método com parâmetros, o compilador manda "você" para onde o método está localizado junto com os parâmetros. Depois de executado o método, retorna para o local onde estava. Observe que existem dois passos intermediários, ir até onde o método está e depois retornar, o que consome tempo de processamento.

Para reduzir este problema foram criados os métodos inline. Quando um método inline é chamado, o compilador em vez de passar o endereço do método, coloca uma cópia do mesmo onde ele está sendo chamado. É o mesmo que ocorria com as antigas macros de C, com a vantagem de fazer verificação de tipo.

Exemplo:

```

//-----A.h
class A
{
//inline explícito, uso da palavra chave inline
inline int funçãoA(int a, int b)
    {return a*b;};
//inline implícito, pois o método é definido dentro da classe
int funçãoB(int aa, int bb)
    {return aa/bb;};
//vai ser definida no arquivo A.cpp (é inline)
inline void funçãoC();
};
//-----A.cpp
//O método é inline, porque foi definido assim dentro da classe A
void A::funçãoC() {};

```

Sentenças para métodos inline

- São métodos pequenos, daí o termo em linha.
- São métodos que são executados mais rapidamente.
- Uma especificação inline é uma sugestão para que seja feita a sua substituição e não a sua chamada, o compilador é que resolve.

- Se o método for grande, o compilador vai desconsiderar o especificador inline.
- O uso de métodos inline torna o código mais rápido, porém maior.
- inline é ideal para retorno de valores.
- Um método definido dentro da declaração da classe é inline por default.
- Uma método recursivo (que chama a si mesmo) não pode ser inline.
- Alterações em métodos inline implicam na necessidade de se recompilar todas as bibliotecas que façam uso do mesmo, ou seja, use inline com cuidado.

Veja na Figura 10.2 o diagrama UML da classe TPonto. A listagem do código para implementar a classe TPonto é ilustrada a seguir. Observe o uso de métodos inline para acesso aos atributos da classe. O uso de this será explicado posteriormente. A classe TPonto será utilizada posteriormente.

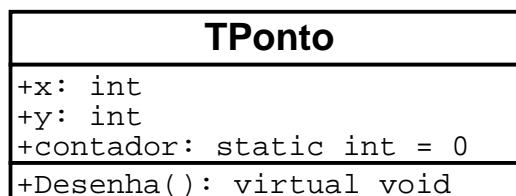


Figura 10.2: A classe TPonto.

Listing 10.5: Arquivo e87-TPonto.h.

```
//-----Arquivo e87-TPonto.h
#ifndef _TPonto_
#define _TPonto_

/*
Define a classe TPonto
Define o tipo de usuário TPonto.
*/
class TPonto
{
//-----Atributos
//controle de acesso
protected:

//atributos de objeto
int x;
int y;

//atributo de classe
static int contador;

//-----Métodos
public:

//Construtor default
```

```

TPonto():x(0),y(0)
    {contador++;};

//Construtor sobrecarregado
TPonto(int _x,int _y):x(_x),y(_y)
    {contador++;};

//Construtor de cópia
TPonto(const TPonto& p)
    {
        x = p.x;
        y = p.y;
        contador++ ;
    };

//Destrutor virtual
virtual ~TPonto()
    {contador--;};

//Seta ponto
inline void Set(TPonto& p);

//Seta ponto
inline void Set(int & x, int & y);

//Método inline definido dentro da classe
int Getx() const { return x; };

//Método inline, declarado aqui, definido no arquivo cpp
inline int Gety() const;

//Método virtual, desenha o ponto
virtual void Desenha();

//Método Estático e const
static int GetContador();
};

#endif

```

Listing 10.6: Arquivo e87-TPonto.cpp.

```

//-----Arquivo TPonto.cpp
#include <iostream>
#include "e87-TPonto.h"

//Atributos estáticos da classe devem ser definidos da seguinte forma
//tipo nomeclasse::nomeatributo = valor;
int TPonto::contador = 0;

//Definição dos métodos de TPonto
void TPonto::Set(TPonto& p)
    {
        x = p.Getx();
        y = p.Gety();
    }

```

```

void TPonto::Set(int & _x, int & _y)
{
    x = _x;
    y = _y;
}

int TPonto::Getx() const
{
    return x;
}

int TPonto::GetContador()
{
    return contador;
}

void TPonto::Desenha()
{
    std::cout << "\nTPonto:□Coordenada□x=" << x;
    std::cout << "\nTPonto:□Coordenada□y=" << y <<std::endl;
}

```

Listing 10.7: Uso de métodos e atributos de uma classe.

```

//-----Arquivo prog.cpp
#include <iostream>
#include "e87-TPonto.h"

//Exemplo de criação e uso do objeto TPonto
int main()
{
    int x = 5;
    int y = 4;
    {
        //Cria objeto do tipo TPonto com nome ponto
        TPonto ponto;

        //Chama método Set do objeto ponto
        ponto.Set(x,y);
        ponto.Desenha();
    } //sai de escopo e detroe o objeto ponto

    //chama método estático, observe que não precisa de um objeto
    cout<< "Contador□=□"<<TPonto::GetContador()<<endl;
}

/*
Saída:
-----
[andre@mercurio Cap3-P00UsandoC++]$ ./a.out

Ponto: Coordenada x=5
Ponto: Coordenada y=4

```

```
Contador = 0
*/
```

10.8 Sentenças para métodos

- Os métodos públicos formam a interface da classe e devem ter nomes claros e uniformes.
- Se um método qualquer receber um objeto `const`, este método só poderá acessar os métodos `const` deste objeto, ou seja, não poderá acessar os métodos que modificam o estado do objeto.
- Se um parâmetro de um método não é modificado dentro do método, transforme o mesmo em parâmetro `const`.
- Se um método não altera os atributos do objeto, declare o mesmo como `const`.
- Se um método manipula preferencialmente os atributos de um objeto, ele provavelmente é um método deste objeto.
- ² Um método membro estático não recebe um ponteiro `this`, ou seja, não sabe qual objeto a esta acessando.
- ² Para possibilitar a chamada de métodos encadeados, o método deve retornar um objeto (um `lvalue`).

Exemplo:

```
tipo& F1()
{return *this;}
//Uso
( (obj.F1() ).F2() ).F3();
```

Capítulo 11

Sobrecarga de Métodos

Neste capítulo vamos apresentar a sobrecarga de métodos, o uso de métodos com o mesmo nome mas parâmetros diferentes.

11.1 O que é a sobrecarga de métodos ?

Sobrecarga de métodos se refere ao uso de métodos com mesmo nome, mas com tipos de parâmetros ou número de parâmetros diferentes. Isto é, o nome do método é o mesmo mas os tipos de parâmetros são diferentes. De um modo geral como os métodos sobrecarregados tem o mesmo nome, eles realizam basicamente a mesma tarefa, a diferença é o número de parâmetros e ou o tipo dos parâmetros que são recebidos.

11.2 Exemplos de sobrecarga

O exemplo a seguir declara métodos com o mesmo nome, métodos sobrecarregados.

```
Exemplo:  
void funçãoA(int x, int y);      //1-int,int  
void funçãoA(float x, float y); //2-float,float  
void funçãoA(int x, float y);   //3-int,float
```

O compilador reconhece qual método você quer acessar verificando o tipo dos parâmetros e o número de parâmetros.

Abaixo, usa a funçãoA, o compilador identifica que x e y são do tipo int e chama a primeira função declarada.

```
Exemplo:  
int x = 3; int y = 4;  
funçãoA(x,y);           //Acessa funçãoA //1
```

Observe que mudar o nome dos parâmetros não é uma sobrecarga, o compilador diferencia o tipo e não o nome.

```
Exemplo  
int funçãoA(int z, int r);      //4-erro já declarada
```

No exemplo acima ocorre um erro, pois tem como parâmetros dois inteiros, repetindo a declaração `void funçãoA(int x,int y);`. Ou seja, você deve estar atento as conversões de tipo quando declara métodos sobrecarregados. Tome os cuidados abaixo:

- O tipo e a referência para o tipo.

```
f(int a);  
f(int & a); //erro, redeclaração
```

- Somente a diferenciação do nome dos parâmetros não é sobrecarga.

```
f(int a, int b);  
f(int c, int d); //erro, redeclaração
```

- Um método com parâmetros default é uma sobrecarga:

```
void jogo(int a ,int b ,int c = 1);  
//cria os dois métodos abaixo  
//void jogo(int a ,int b ,int c = 1);  
//void jogo(int a ,int b);
```

- Os valores de retorno não são avaliados em uma sobrecarga.
- Se o método sobrecarregado recebe int, faça uma análise das possíveis conversões automáticas.

Capítulo 12

Uso de Ponteiros e Referências

Neste capítulo apresenta-se os ponteiros, os ponteiros const e a conversão de ponteiros. Exemplos de uso de ponteiros e classes, o ponteiro this, o uso de ponteiros para criar e usar objetos dinâmicos, a forma de uso de ponteiros para atributos e métodos de uma classe. No final do capítulo apresenta-se o uso de referências.

12.1 Ponteiros

Ponteiros são objetos cujo conteúdo é o endereço de outros objetos. É um objeto com o endereço de outro objeto.

Na prática os ponteiros são usados para substituir os objetos originais. Sua vantagem está associada ao seu pequeno tamanho. Assim, passar um ponteiro de um objeto para um método é mais rápido e econômico que passar uma cópia do objeto.

Para declarar ponteiros precedemos o nome do objeto pelo asterisco *, ou seja, para um tipo T, T* é um ponteiro.

O procedimento de uso dos ponteiros tem 3 etapas. Primeiro cria-se o ponteiro.

```
tipo * ponteiro = NULL;
```

A seguir coloca-se no ponteiro o endereço do objeto para o qual ele vai apontar.

```
tipo objeto;  
ponteiro=& objeto;
```

Finalmente, utiliza-se o ponteiro.

```
*ponteiro = algo; //armazena algo no objeto
```

Vamos agora tentar explicar o funcionamento e o uso dos ponteiros.

Digamos que o sistema de armazenamento de objetos em um programa tenha um carteiro. Isto é, o carteiro deve pegar um valor e armazenar no objeto. Para realizar este trabalho o carteiro precisa do valor (ou seja, um pacote para entregar) e do endereço do objeto onde o valor deve ser armazenado (endereço de entrega).

Quando você faz `x=5`; está dizendo para o carteiro pegar o valor 5 e levar até a casa de x. Um ponteiro pode ser imaginado como um endereçador indireto para o carteiro. Veja o exemplo a seguir, o mesmo é ilustrado na Figura 12.1.

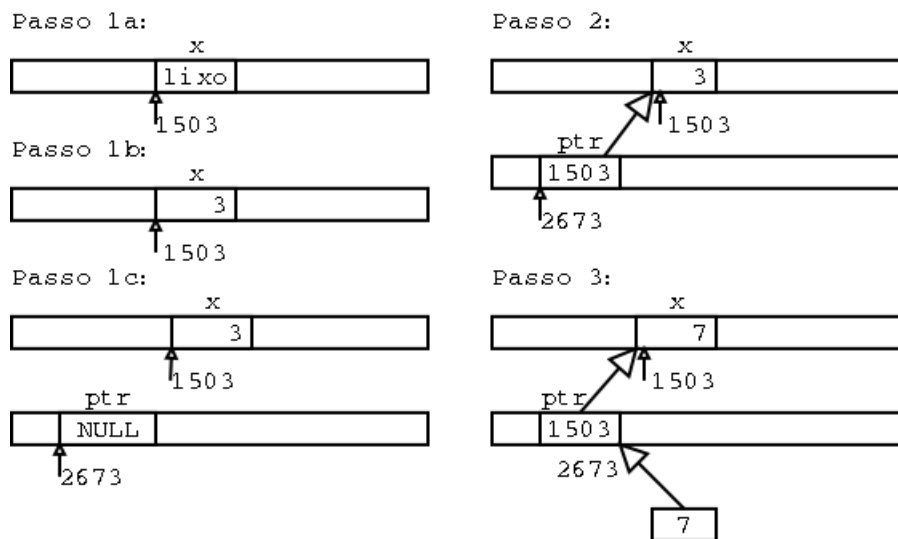


Figura 12.1: Como declarar e usar um ponteiro.

Exemplo:

```
//Passo 1a
int x;
//Cria objeto do tipo int, com nome x
//o endereço do objeto x é sua posição na memória do computador
//admita que o endereço de x é =1503
//Acima constrõe uma casa para x, no endereço 1503,
//Passo 1b
x = 3;
//Armazena em x o valor 3
//Acima diz para o carteiro levar o valor 3, até a casa de x (no número 1503)
//Passo 1c
int* ptr = NULL;
//acima constrõe um objeto ponteiro, do tipo int*, com nome ptr
//conteúdo de ptr=NULL
//Passo 2
ptr = &x;
//pega o endereço de x (valor 1503) e armazena na casa de ptr logo ptr = 1503
//Passo 3
*ptr = 7;
//o carteiro pega o valor 7, vai até a casa de ptr,
//chegando lá, ele encontra o endereço de x, e recebe
//instrução para levar até x. A seguir leva o valor 7 até x.
```

Observe no exemplo acima os 3 passos:

1. Declaração do ponteiro

```
int* ptr = NULL;
```


2. Colocação do endereço do objeto no ponteiro

```
int x = 3;
ptr  = &x;
```

3. Uso do ponteiro

```
*ptr = 7;
```

12.2 Criação e uso de objetos dinâmicos com ponteiros

Os ponteiros são usados para criar, usar e deletar objetos dinamicamente. Mas porque devo usar objetos dinâmicos ?

12.2.1 Porque usar objetos dinâmicos ?

O uso de objetos dinâmicos possibilita, por parte do programador, um controle mais eficiente da memória utilizada. Quando os objetos são alocados em tempo de compilação, o programador deve definir o tamanho dos objetos, assim, uma string de C precisa ter seu tamanho previamente definido, veja o exemplo:

```
char nomePessoa[50];
```

No exemplo acima, cria um vetor de C para armazenar uma string com 50 caracteres.

Mas se o que acontece se nome tiver mais de 50 caracteres ?. Vai ocorrer um estouro de pilha e o programa vai travar.

E se o nome tiver apenas 20 caracteres ?. Então você estará desperdiçando 30 caracteres.

Para evitar o estouro de pilha e o desperdício de memória, utiliza-se os objetos dinâmicos. Veja o exemplo.

```
char * nomePessoa = new char [tamanhoNecessário];
```

O operador new

Observe o uso do operador new. O operador new é utilizado para alocar um bloco de memória. Primeiro new solicita ao sistema operacional um bloco de memória, após alocar o bloco de memória, new retorna um ponteiro para o bloco alocado. Se new falhar, retorna um bad_alloc.

O operador delete

Para destruir o objeto e devolver a memória para o sistema operacional utiliza-se o operador delete, delete destrói o objeto e devolve o bloco de memória para o sistema operacional.

```
delete nomePessoa;
```

Observe que apenas o bloco de memória é destruído. O ponteiro continua existindo, e continua apontando para o bloco de memória que agora não existe mais. Ou seja, depois de usar delete ponteiro, o ponteiro aponta para um monte de lixo e não deve mais ser utilizado.

Os operadores new e delete são detalhados no Capítulo sobre operadores.

Veja no exemplo a seguir a utilização de objetos dinâmicos com ponteiros. A listagem inicia com a inclusão da biblioteca <iostream> para entrada e saída de dados. A seguir inclui a classe TPonto que definimos em listagem anterior. Cria um ponteiro para TPonto (TPonto*) e aloca o objeto com new. Depois de criado o objeto dinâmico, o mesmo é usado. Compare este exemplo com o apresentado na listagem 10.7, lá usava métodos normais para criar o objeto TPonto, aqui usa mecanismos dinâmicos. Veja que a forma de uso do objeto muda. Em objetos normais utiliza-se NomeObjeto.atributo e em objetos dinâmicos NomeObjeto->atributo.

Listing 12.1: Usando ponteiro para criar e usar objetos dinâmicos.

```
//----- Arquivo e84-ProgTPontoDinamico.cpp
#include <iostream>
#include "e87-TPonto.h"

//Exemplo de criação e uso do objeto TPonto
int main()
{
    int x = 5;
    int y = 4;

    //Cria ponteiro para TPonto
    TPonto* ptr = NULL;

    //Cria objeto do tipo TPonto, e coloca endereço em ptr
    //O operador new é usado para criar um objeto novo.
    //new retorna um ponteiro para o objeto criado.
    //se new falha (por falta de memória em seu micro), retorna um bad_alloc.
    ptr = new TPonto;

    //Chama método Set do objeto ptr
    x = 6; y = 7;
    ptr->Set(x,y);
    ptr->Desenha();
    int xx = ptr->Getx();
}

/*
Novidade:
-----
-Uso de ponteiro e objeto dinâmico
-Uso do operador new
-Observação:
Observe que com objetos estáticos usa-se nomeobjeto.atributo;
e com objeto dinâmicos, troca o ponto (.) pela seta (->)
numerobjeto->atributo;
*/

/*
Saída:
```

```

-----
[andre@mercurio Cap3-P00UsandoC++]$ ./a.out
TPonto: Coordenada x=6
TPonto: Coordenada y=7
Contador = 1
*/

```

12.2.2 Controle da criação e deleção de objetos com ponteiros²

Você pode adotar o tipo de controle, abordado no exemplo abaixo.

```

Exemplo:
//Cria o ponteiro e zera
TObj *ptr = NULL;
....
//Abaixo, se ptr for NULL, delete ptr não tem efeito,
//se ptr aponta para um objeto, destrói o objeto
delete ptr;
//depois de deletar sempre faça ptr = NULL;
ptr = NULL;
//Cria objeto novo e armazena endereço em ptr
ptr = new TObj();
if( ptr == NULL)
{
    cout<<"\nobjeto não alocado"<<endl;
    exit();
}
//Usa o ponteiro
ptr->Funcao()
...
//deleta ao final,
//depois de deletar sempre aponta para NULL
delete ptr;
ptr = NULL;

```

Neste tipo de controle, nunca ocorre estouro por deleção de ponteiro, pois o ponteiro é sempre NULL ou aponta para um objeto válido. Observe que nunca testa antes de deletar, mas sempre faz ptr= NULL na hora de criar e após deletar.

12.3 Ponteiros const e ponteiros para const

A palavra chave *const* é usada para informar que o objeto é constante, não muda. A mesma pode ser usada com ponteiros, significando que o ponteiro aponta sempre para o mesmo local de memória, ou que o objeto apontado não muda, ou ambos. Entenda abaixo os diferentes formatos de uso de const com ponteiros.

Ponteiro para uma constante: O conteúdo do objeto apontado é constante, mas o ponteiro pode ser apontado para outro objeto.

Exemplo:

```
const float *ptr;
int a,b;
ptr = & a ;           //ok
ptr = & b ;           //ok
*ptr = 6 ;            //erro, conteúdo constante, não pode modificar
```

Ponteiro constante: Quando o ponteiro aponta para o mesmo local da memória, o objeto apontado pode ser alterado.

Exemplo:

```
int a,b;
float* const ptr = &a; //ok (cria e inicializa)
ptr = & b ;           //erro, aponta sempre para mesmo objeto (endereço)
*ptr = 6 ;            //ok
```

Ponteiro constante para objeto constante: Neste caso tanto o ponteiro como o objeto apontado não podem ser alterados.

Exemplo:

```
double dpi = 3.141516;
const double * const pi = &dpi;
double d = 3.3;
ptr = & d ;           //erro, ptr aponta sempre para mesmo objeto (endereço)
*ptr = 6.2 ;         //erro, o conteúdo do objeto apontado não pode mudar.
```

12.4 Conversão de ponteiros²

A Tabela 12.1 mostra a conversão de ponteiros. Na primeira linha um Tipo é convertido em uma referência para o tipo (Tipo&). A compreensão desta tabela é importante, leia cada linha com cuidado.

Tabela 12.1: Conversão de ponteiros objetos.

Dê	Para
Tipo	Tipo&
Tipo&	Tipo
Tipo[]	Tipo *
Tipo(argumentos)	Tipo(*) (argumentos)
Tipo	const Tipo
Tipo	volatile Tipo
Tipo*	const Tipo*
Tipo*	volatile Tipo*

12.5 Ponteiro this

Na parte de análise orientada a objeto, vimos que a classe é definida como uma fábrica de objetos, e que é a classe que define a forma do objeto. Vimos ainda que quando se cria um objeto é reservado espaço na memória para inclusão de todos os atributos não estáticos do objeto, e que não é reservado espaço para os métodos. Assim, um objeto na memória do computador contém somente atributos. Os métodos não são criados para cada objeto, eles ficam armazenados na classe.

Isto faz sentido, pois dois objetos da mesma classe terão atributos diferentes, mas os métodos serão os mesmos, isto é, quando um objeto acessa um de seus métodos, ele está acessando os métodos da classe.

Como os métodos são os mesmos para todos os objetos da classe é necessário um dispositivo de identificação de qual objeto está acessando o método. Este dispositivo é implementado através do ponteiro `this`.

Através do ponteiro `this` o método da classe sabe qual objeto está acessando, ou seja, *this é um ponteiro para o objeto que é passado implicitamente para o método.*

```
Exemplo:
//contador é um atributo do objeto
NomeClasse NomeClasse::operator++()
{
    this->contador++;
    return *this
}
```

O compilador traduz a sua chamada de método da seguinte forma:

```
ptr_objeto->função(parâmetros); //C++
função(ptr_objeto,parâmetros); //Tradução para C
```

Resumo: Você já deve ter se perguntado, como é que um método da classe acessa o atributo `x` do objeto1 e não do objeto2 ? É que quando um objeto chama um método da classe, este passa para o método o seu endereço através do ponteiro `this`. Desta forma, ao usar um atributo `x`, na realidade o método está usando **this->x**; Assim, **this** é um ponteiro que é passado implicitamente a um método da classe, informando qual objeto está acessando.

```
this=& objeto; //this contém o endereço do objeto.
```

12.5.1 Sentenças para ponteiro this

- ² Um ponteiro `this` da classe `X` é do tipo `X* const`, isto é, aponta sempre para o mesmo objeto.

12.6 Usando `auto_ptr`²

Como descrito no capítulo de exceções, se você tem um objeto dinâmico [Ex: `int* ptr = new int(30);`], e ocorre uma exceção, você deve prover mecanismos para deletar os objetos dinâmicos. O que pode ser complicado e/ou chato.

O `auto_ptr` é uma classe ponteiro que tem uma relação íntima com a RTTI, de forma que, se ocorrer uma excessão após a alocação de um conjunto de objetos dinâmicos, os mesmos se auto deletam. Ou seja, ao sair de escopo um ponteiro `auto_ptr` automaticamente chama o destrutor do objeto.

A classe `auto_ptr` é definida em `<memory>`. Veja o exemplo.

Listing 12.2: Comparando o uso de vetores estáticos de C, dinâmicos de C++, com `auto_ptr` de C++ e `vector` da `stl`.

```
//-----e74-auto-ptr.cpp
#include <iostream>
#include <memory>
#include <vector>
using namespace std;

class Tipo
{
public:
    int t;
    static int cont;

    Tipo ( )
        { cont++;
          cout << "Construtor do objeto, cont=" << cont << endl ;
        };
    ~Tipo()
        { cout << "Destrutor do objeto, com cont=" << cont << endl ;
          cont--;
        }
};

int Tipo::cont = 0 ;

int main()
{
    cout<<"-----vetor estático de C:"<<endl;
    {
        Tipo v_static[2];                //cria vetor estático
    }                                    //destrõe vetor ao sair de escopo

    cout<<"-----vetor dinâmico em C++ sem STL:"<<endl;
    {
        Tipo* v_dinamico = new Tipo [3];
        //.....usa o vetor...
        delete []v_dinamico;            //precisa do delete []
    }

    //Usando auto_ptr (criar apenas um objeto)
    //auto_ptr não deve apontar para um vetor,
    cout<<"-----Objeto dinâmico em C++ com auto_ptr:"<<endl;
    {
        auto_ptr<Tipo> v_autoptr(new Tipo);
        v_autoptr->t = 77;
        cout << "t=" <<v_autoptr->t << endl;
        //.....usa o vetor...
    }
}
```

```

} //deletado automaticamente

cout<<"-----vetor dinâmico em C++ com STL:"<<endl;
{
    vector< Tipo > v_stl(4,Tipo()); //é dinâmico
    for(int i = 0 ; i < v_stl.size() ; i++)
    {
        v_stl[i].t = i;
        cout << i << " = " <<v_stl[i].t << endl;
    }
} //destrõe objeto ao sair de escopo

Tipo::cont = 0 ;
cout<<"-----vetor de ponteiros em C++ com STL:"<<endl;
{
    vector< Tipo* > v_stl(5);
    for(int i = 0 ; i < v_stl.size() ; i++)
    {
        v_stl[i] = new Tipo();
        v_stl[i]->t = i;
        cout << "i="<< i << " t=" <<v_stl[i]->t << endl;
    }
    for(int i = 0 ; i < v_stl.size() ; i++) delete v_stl[i] ;
}
}

/*
Saída:
-----
[andre@mercurio Cap2-Sintaxe]$ ./a.out
-----vetor estático de C:
Construtor do objeto, cont = 1
Construtor do objeto, cont = 2
Destrutor do objeto, com cont =2
Destrutor do objeto, com cont =1
-----vetor dinâmico em C++ sem STL:
Construtor do objeto, cont = 1
Construtor do objeto, cont = 2
Construtor do objeto, cont = 3
Destrutor do objeto, com cont =3
Destrutor do objeto, com cont =2
Destrutor do objeto, com cont =1
-----Objeto dinâmico em C++ com auto_ptr:
Construtor cont = 1
t = 77
Destrutor do objeto, com cont =1
-----vetor dinâmico em C++ com STL:
Construtor do objeto, cont = 1
Destrutor do objeto, com cont =1
0 = 0 1 = 1 2 = 2 3 = 3
Destrutor do objeto, com cont =0
Destrutor do objeto, com cont =-1
Destrutor do objeto, com cont =-2
Destrutor do objeto, com cont =-3 <-AQUI
-----vetor de ponteiros em C++ com STL:

```

```

Construtor do objeto, cont = 1
i=0 t= 0
Construtor do objeto, cont = 2
i=1 t= 1
Construtor do objeto, cont = 3
i=2 t= 2
Construtor do objeto, cont = 4
i=3 t= 3
Construtor do objeto, cont = 5
i=4 t= 4
Destrutor do objeto, com cont =5
Destrutor do objeto, com cont =4
Destrutor do objeto, com cont =3
Destrutor do objeto, com cont =2
Destrutor do objeto, com cont =1
*/
/*
Observação:
No exemplo de uso de vector dinâmico da stl, esta criando
o objeto uma única vez (visto que o construtor só é executado uma vez).
Mas esta deletando o objeto 4 vezes.
Como proceder para corrigir este problema ?
*/

```

Observe que o mais fácil de usar e mais versátil é o vector da STL.

12.7 Ponteiros para métodos e atributos da classe³

Este é um título de nível 3, isto significa que só deve ser lido por usuário experiente.

Em algum caso, muitíssimo especial (raro), você pode querer ter um ponteiro para um atributo ou método de uma classe, esta seção mostra, através de um exemplo, como você deve proceder.

Exemplo:

```

class A
{
int x;      //Atributo
void fx(); //Método
}
//Criando ponteiro para função fx() da classe A
void(A::*ptrFuncao)() = & A::fx();
//Ponteiro para atributo x da classe A
int A::*ptr_x = & A::x;
cout << (*ptr_x) << endl;

```

12.8 Sentenças para ponteiros

- Se você deseja acessar um método de um objeto dinâmico, pode usar uma das duas opções:

```

Exemplo:
ptr->função();

```



```
*ptr.função();
```

12.9 Referências (&)

Uma referência pode ser encarada como um outro nome para um objeto.

Uma referência deve ser definida uma única vez, assim, uma referência aponta sempre para o mesmo local da memória.

```
Exemplo:
int v1 = 5;
int v2;
//declara uma referência a v1
//ou seja, ref_v1 é o mesmo que v1
int& ref_v1 = v1;
//Para armazenar v1 em v2
v2 = ref_v1;
//Para armazenar algo em v1 usando a referência
ref_v1 = algo;
//Pegar o endereço da referência é o mesmo que pegar o endereço do
//objeto referenciado. Armazena endereço de v1 em ptr
int* ptr;
ptr = & ref_v1;
```

12.9.1 Diferenças entre referência e ponteiro

Uma referência não reserva espaço na memória para si próprio, ao contrário dos ponteiros que reservam um espaço na memória.

Observe que existe ponteiro de ponteiro, mas não existe referência de referência.

Listing 12.3: Uso de referência.

```
#include <iostream>

//O comando using, informa que voce vai usar o objeto
//std::cout,
//desta forma, em vez de digitar
// std::cout<<"Entre com x:";
//voce pode digitar
// cout<<"Entre com x:";
using std::cout;
using std::endl;

int main()
{
    //tipo=int, nome=x, valor=3
    int x = 3;

    //tipo= referência para inteiro, nome=ref, valor=x
    //Daqui para frente, ref é a mesma coisa que x.
    int& ref = x;
```

```

    cout << "Valor de x=" << x << endl << "Valor da ref=" << ref << endl;

    ref = 156;
    cout << "Mudou ref" << endl;
    cout << "Valor de x=" << x << endl << "Valor da ref=" << ref << endl;

    x = 6;
    cout << "Mudou x" << endl;
    cout << "Valor de x=" << x << endl << "Valor da ref=" << ref << endl;
    return 0;
}

/*
Novidade:
-----
Uso de Referência.
Uma referencia é um outro nome para um objeto. No exemplo acima, ref é um outro
nome para x.
As referências são usadas principalmente como argumentos de uma função.
*/

/*
Saída:
-----
[andre@mercurio Cap2-Sintaxe]$ ./a.out
Valor de x = 3
Valor da ref = 3
Mudou ref
Valor de x = 156
Valor da ref = 156
Mudou x
Valor de x = 6
Valor da ref = 6
*/

```

12.9.2 Referências para ponteiros²

Você pode declarar uma referência para um ponteiro, ou seja, um outro nome para um ponteiro.

Exemplo:

```

int * ponteiro;
tipo * & referencia_de_ponteiro = ponteiro;

```

Referências para ponteiros costumam ser usadas como parâmetros de métodos.

12.9.3 Sentenças para referências

- Uma referência não pode ser alterada para referenciar outro objeto após sua inicialização (ou seja uma referência se comporta como um ponteiro const).
- Como as referências não são objetos não podem existir matrizes de referências.

- Novamente, o maior uso de referências para ponteiros ocorre como parâmetro de métodos.
- Se uma função espera uma referência e recebe um ponteiro ela aceita mas pode causar um bug.

Exemplo:

```
//Prototipo do método:  
//O método espera uma referência  
TBitmap(const TDC& dc, const TDib& dib);  
TDib* dib;  
TBitmap (dc,dib); //Uso errado, passando um ponteiro  
TBitmap (dc,*dib); //uso correto, passando o objeto
```

- ²BUG: Digamos que você deseja passar um objeto como parâmetro para um método f(nomeclasse obj). Como você está passando o objeto por cópia, vai criar uma cópia do objeto. Depois vai usar dentro do método, e, ao encerrar o método, o objeto é deletado, chamando o método destrutor. Se o objeto original tinha algum ponteiro com alocação dinâmica, este ponteiro é deletado, sendo deletado o bloco de memória por ele acessado. Assim, o ponteiro do objeto original aponta agora para um monte de lixo. Para que isto não ocorra você deve passar uma referência do objeto, de forma que é passado o objeto e não uma cópia deste, quando sair de escopo o objeto não é eliminado.

Capítulo 13

Métodos Construtores e Destrutores

Apresenta-se neste capítulo os métodos construtores, incluindo o construtor default e o construtor de cópia, a seguir, apresenta-se os métodos destrutores e a ordem de criação e de destruição dos objetos.

13.1 Protótipo para construtores e destrutores

Apresenta-se a seguir o protótipo para declaração dos métodos construtores e destrutores. Ao lado do protótipo o número da seção onde o mesmo é discutido. Observe que o construtor tem o mesmo nome da classe e não retorna nada, nem mesmo void. O destrutor tem o mesmo nome da classe precedido do til (~).

Protótipo:

```
class CNome
{
    CNome(); //Construtor default, seção 13.3
    CNome(parâmetros); //Construtor sobrecarregado, seção 13.3
    CNome(const CNome& obj); //Construtor de cópia, seção 13.4
    CNome(const CNome& obj, int=0); //Construtor de cópia, seção 13.4
    Tipo(); //Construtor de conversão, seção 20.3
    ~CNome(); //Métodos destrutores, seção 13.5
    virtual ~CNome(); //Destrutor virtual, seção 16.2
};
Class CNomeDer:public CNome
{
    operator CNome(); //construtor de conversão em herança, seção ??
}
```

13.2 Métodos construtores

Um método construtor é um método como outro qualquer, com a diferença de ser automaticamente executado quando o objeto é criado. O objetivo dos métodos construtores é inicializar os atributos do objeto, ou seja, definir os valores iniciais dos atributos do objeto.

Quando você cria um objeto (ex: CNome objeto;), a sequência de construção do objeto é dada por:

1. Solicitação de memória para o sistema operacional.
2. Criação dos atributos do objetos.
3. Execução do construtor da classe.

Veja o exemplo a seguir.

Exemplo:

```
//-----Arquivo X.h
class X
{public:
//declara atributos a,b,c
int a,b,c;
//declaração do construtor
X(int i, int j);
};
//-----Arquivo X.cpp
//definição do construtor define os valores dos atributos
//Preste atenção no formato
X::X(int i, int j): a(i), b(j),c(0)
{
//No exemplo acima, inicializa o valor a com i
//(o mesmo que a = i), b com j e c com 0.
//sequência do construtor...
};
```

Dentro do método construtor, você pode, além de inicializar os atributos do objeto, realizar outras tarefas para inicializar o objeto. Por exemplo, se sua classe representa uma impressora, você pode verificar se existe uma impressora conectada ao seu micro.

13.2.1 Sentenças para construtores

- Tem o mesmo nome da classe.
- Não deve retornar nenhum tipo de valor, nem mesmo void.
- Os construtores podem ser sobrecarregados.
- Não podem ser virtuais.
- São sempre públicos.
- Pode-se inicializar os atributos do objeto no construtor.
- Crie variáveis dinâmicas no construtor com new a apague no destrutor com delete.
- Não podem ser const nem volátil.

13.3 Construtor default

Se você não criar um método construtor, o compilador cria um construtor vazio, que não recebe nenhum argumento e é chamado de *construtor default*.

Se você criar um construtor, deixa de existir o construtor default.

Exemplo:

```
class TNomeClasse
{
    int a;
    //O compilador automaticamente cria o
    //método construtor abaixo, sem parâmetros e vazio
    //TNomeClasse(){};
};
//Criando 30 objetos
//na linha abaixo usa o construtor default da classe
TNomeClasse vetorEstatico [30];
```

13.3.1 Sentenças para construtor default

- Se na classe existirem referências ou const, o compilador não cria o construtor default. Neste caso, você vai ter de criar o construtor default. Isto ocorre porque você precisa inicializar os objetos const e as referências.
- Sempre crie um construtor default, assim você evita problemas.
- Ao criar um vetor estático de objetos, você é obrigado a usar o construtor default, para usar um outro construtor você vai ter de criar os objetos um a um dinamicamente.
- Em determinados casos é conveniente termos um método de inicialização, um método usado para inicializar os atributos da classe, e que pode ser chamado a qualquer momento para reinicializar os atributos com valores padrões.
- Membros estáticos da classe devem ser definidos fora da classe.
- ³Tipos inteiros constantes e estáticos podem ser definidos dentro da classe.

Exemplo:

```
static const int valor = 5;
```

13.4 Construtor de cópia X(const X& obj)

O construtor de cópia é usado para criar uma cópia de um objeto existente. O construtor de cópia recebe como parâmetro um objeto da própria classe.

Um construtor de cópia é criado automaticamente pelo compilador quando você usa o operador = (igual).

Exemplo:

```
//cria objeto p1, usa o construtor default
TPonto p1;
//cria objeto p2, usa o construtor de cópia,
//os atributos de p2 serão iguais aos de p1
TPonto p2 = p1;
//cria objeto p3, usa construtor de cópia
//Atributos de p3 serão iguais aos de p1
TPonto p3(p1);
//usa construtor default
TPonto p4();
//Abaixo usa o operador =
p4 = p3;
//Abaixo usa operador ==
if(p4 == p2)
{...}
```

Veja na Figura 13.1 a classe TAluno.

TAluno
+nome: string
+matricula: string
+iaa: float
+numeroAlunos: static int = 0
+Entrada(): void
+Saida(): void

Figura 13.1: A classe TAluno.

Na listagem a seguir um exemplo com construtor default e construtor de cópia.

Listing 13.1: Uso de construtor default e de copia.

```
//-----Arquivo TAluno
.h
//-----Bibliotecas C/
C++
#include <iostream>
#include <string>
#include <vector>

using namespace std;

//-----Classe
/*
A classe TPessoa representa uma pessoa (um aluno ou um professor)
de uma universidade.
Tem um nome, uma matricula e um IAA.
E métodos básicos para entrada e saída de dados.
Inclue construtor e destrutor (declarados e definidos dentro da classe).

```



```

*/
class TPessoa
{
//-----Atributos
//Acesso privado
private:

    //Atributo normal é criado para cada objeto
    string nome;
    string matricula;
    double iaa;

    //Atributo estático é criado na classe (aqui é a declaração)
    static int numeroAlunos;

//-----Métodos
    Construtores

//Acesso público (tendo um objeto pode acessar os métodos publicos)
public:

    //Construtor default
    //Chamado automaticamente na construção do objeto
    //observe a forma de inicialização do atributo iaa.
    TPessoa() : iaa(0)
    {
        numeroAlunos++;
        cout<<"criou objeto " << numeroAlunos << " construtor default " <<
            endl;
    };

    //Construtor de cópia
    //Cria uma cópia de um objeto existente observe que cada atributo é copiado
    TPessoa(const TPessoa& obj)
    {
        nome = obj.nome;
        matricula = obj.matricula;
        iaa = obj.iaa;
        numeroAlunos++;
        cout<<"criou objeto " << numeroAlunos << " construtor de cópia " <<
            endl;
    }

//-----Método
    Destrutor
    //Chamada automaticamente na destruição do objeto
    //Só existe um destrutor para cada classe
    ~TPessoa()
    {
        numeroAlunos--;
        cout<<"destruiu objeto " << numeroAlunos << endl; //opcional
    };

//-----Métodos

```

```

//Métodos do objeto, alteram as propriedades do objeto
//Leitura dos atributos (nome, matricula, iaa)
void Entrada();

//Saida dos atributos (nome, matricula, iaa)
void Saida(ostream &os) const;

//Métodos Get
string Getnome()      const   {return nome;};
string Getmatricula() const   {return matricula;};
double Getiaa()       const   {return iaa;};

//Métodos Set
double Setiaa(double _iaa)          {iaa=_iaa;}
void Setnome(string _nome)          {nome=_nome;}
void Setmatricula(string _m)        {matricula=_m;}

//-----Métodos
    Estáticos
//Métodos static podem ser chamados sem um objeto
//e só podem manipular atributos static
    static int GetnumeroAlunos(){return numeroAlunos;};
};

/*
//-----Arquivo TAluno
    .cpp
*/
//A linha abaixo define (aloca memória) para o atributo numeroAlunos.
int TPessoa::numeroAlunos=0;

//Definição dos métodos
void TPessoa::Entrada()
{
    cout << "Entre com o nome do aluno: ";
    getline(cin,nome);

    cout << "Entre com a matricula do aluno: ";
    getline(cin,matricula);

    cout << "Entre com o IAA do aluno: ";
    cin>> iaa;
    cin.get();
}

//Método const não altera os atributos do objeto
void TPessoa::Saida(ostream &os) const
{
    os << "Nome do aluno: "          << nome          << endl;
    os << "Matricula: "             << matricula     << endl;
    os << "iaa: "                   << iaa          << endl;
}

//-----Arquivo main.
    cpp

```

```

int main ()
{

    string linha="-----\n
        ";

    //Cria um objeto professor do tipo TPessoa
    TPessoa professor;

    //Compare a entrada abaixo com a de exemplo anterior
    cout << "Entre com o nome do professor: ";
    string nome;
    getline(cin,nome);
    professor.Setnome(nome);

    cout << "Entre com a matricula do professor: ";
    string matricula;
    getline(cin,matricula);
    professor.Setmatricula(matricula);

    cout<<"Entre com o número de alunos da disciplina (ex=3): ";
    int numeroAlunos;
    cin >> numeroAlunos;
    cin.get();

    //Cria um vetor de objetos alunos do tipo TPessoa
    vector< TPessoa > aluno(numeroAlunos);

    for(int contador=0; contador < aluno.size(); contador++)
    {
        cout << "Aluno " << contador << endl;
        aluno[contador].Entrada();
    }

    cout << linha;
    cout << "RELAÇÃO DE PROFESSORES E ALUNOS: " << endl;
    cout << linha;

    cout << "Nome do professor: " << professor.Getnome() << endl;
    cout << "Matricula: " << professor.Getmatricula() << endl;

    for(int contador=0; contador < aluno.size(); contador++)
    {
        cout << linha;
        cout << "Aluno " << contador << endl;
        aluno[contador].Saida(cout);
    }

    //acesso de um método estático (da classe) sem um objeto.
    cout<<"\nNúmero de alunos = " << TPessoa::GetnumeroAlunos() << endl;

    cout << linha;
    cout << linha;
    cout<< " executando: TPessoa professor2(professor); " << endl;
    //uso construtor de copia

```

```

TPessoa professor2(professor);
professor2.Saida(cout);

//Uso construtor de cópia pela atribuição
{
cout << linha;
cout << linha;
cout<< "┘executando┘:TPessoa┘professor3┘=┘professor2;"<<endl;
TPessoa professor3 = professor2;          ///-Cria objeto professor3
professor3.Saida(cout);

//Acessando métodos Get do objeto diretamente
cout << linha;
cout << linha;
cout<<"\nUsando┘métodos┘objeto.Get┘diretamente"<<endl;
cout<<"\np3.Getnome()="<<professor3.Getnome();
cout<<"\np3.Getmatricula()="<<professor3.Getmatricula()<<endl;
}          ///-Destroe professor3

cin.get();
return 0;
}

/*
Novidades:
-----
-Construtor default e de cópia
-Destrutor
-Métodos Get e Set
-Método static para manipular atributo estático
*/
/*
Saída:
-----
//Para compilar no Linux
[andre@mercurio Cap3-P00UsandoC++]$ g++ e92-class-construtor-copia.cpp
//Para rodar o programa no Linux
[andre@mercurio Cap3-P00UsandoC++]$ ./a.out
criou objeto 1 construtor default
Entre com o nome do professor: P.C.Philippi
Entre com a matricula do professor: 1
Entre com o número de alunos da disciplina (ex =3):3
criou objeto 2 construtor default
criou objeto 3 construtor de cópia
criou objeto 4 construtor de cópia
criou objeto 5 construtor de cópia
destruiu objeto 4
Aluno 0
Entre com o nome do aluno: F.S.Magnani
Entre com a matricula do aluno: 2
Entre com o IAA do aluno: 4
Aluno 1
Entre com o nome do aluno: C.P.Fernandes
Entre com a matricula do aluno: 3

```

```

Entre com o IAA do aluno: 3.8
Aluno 2
Entre com o nome do aluno: L.Zhirong
Entre com a matricula do aluno: 4
Entre com o IAA do aluno: 3.9
-----
RELAÇÃO DE PROFESSORES E ALUNOS:
-----
Nome do professor: P.C.Philippi
Matricula : 1
-----
Aluno 0
Nome do aluno: F.S.Magnani
Matricula : 2
iaa : 4
-----
Aluno 1
Nome do aluno: C.P.Fernandes
Matricula : 3
iaa : 3.8
-----
Aluno 2
Nome do aluno: L.Zhirong
Matricula : 4
iaa : 3.9
Número de alunos = 4
-----
    executando :TPessoa professor2(professor);
criou objeto 5 construtor de cópia
Nome do aluno: P.C.Philippi
Matricula : 1
iaa : 0
-----
    executando :TPessoa professor3 = professor2;
criou objeto 6 construtor de cópia
Nome do aluno: P.C.Philippi
Matricula : 1
iaa : 0
-----
-----
Usando métodos objeto.Get diretamente

p3.Getnome()=P.C.Philippi
p3.Getmatricula()=1
destruiu objeto 5

destruiu objeto 4
destruiu objeto 3
destruiu objeto 2
destruiu objeto 1
destruiu objeto 0
*/

```

Se um objeto AA tem atributos dinâmicos, isto é, alocados com new. Ao criar uma cópia do objeto AA com o construtor de cópia, os ponteiros usados para acessar os objetos dinâmicos vão apontar para o mesmo local de memória. Veja o exemplo:

Listing 13.2: Uso indevido do construtor de cópia em objetos com atributos dinâmicos.

```
#include <iostream>

using namespace std;

//-----Arquivo TVetor.h
//-----Classe TVetor
class TVetor
{
public:
int dimensao;
int * ptr_v;

//Método
void Mostra();

//Construtor
TVetor(int n = 10) : dimensao ( n)
{
ptr_v = NULL;
ptr_v = new int (dimensao);
if( ptr_v == NULL )
cout<<"\nFalha alocação"<<endl;
for ( int i = 0 ; i < dimensao; i++)
{
ptr_v[i]= i;
}
};

//Destructor
virtual ~TVetor()
{
delete [] ptr_v;
};
};

//-----Arquivo TVetor.cpp
void TVetor::Mostra()
{
for( int i = 0; i < dimensao ; i++ )
cout <<"□"<< ptr_v[i] << endl;
}

//-----main
void main()
{
TVetor v1(5);
cout<<"Saída de v1.Mostra()"<<endl;
v1.Mostra();
}
```

```

//aqui, v1->dimensao=5, v1->ptr_v = 1232
{
TVetor v2 = v1;
cout<<"Saída de v2.Mostra() após v2=v1"<<endl;
v2.Mostra();
//aqui, v2->dimensao=5, v2->ptr_v = 1232
}

//aqui, v2 foi deletado, pois saiu de escopo
//agora, v1->dimensao=5, v1->ptr_v = 1232
//mas no endereço 1232 não existe mais um objeto.
cout<<"Saída de v1.Mostra() após deleção de v2"<<endl;
v1.Mostra();
}

/*
Saída:
-----
[andre@mercurio Cap3-P00UsandoC++]$ ./a.out
Saída de v1.Mostra()
0
1
2
3
4
Saída de v2.Mostra() após v2 = v1
0
1
2
3
4
Saída de v1.Mostra() após deleção de v2
0
1
2
3
4
Falha de segmentação
*/

/*
Observe que como v2.ptr_v e v1.ptr_v apontavam
para o mesmo bloco de memória, após destruir v2, v1.ptr_v
aponta para um monte de lixo e causa uma falha de segmentação.
*/

```

A solução para este problema é definir manualmente o construtor de cópia, alocando a memória com new para os objetos dinâmicos. Acrescente na classe acima o construtor de cópia dado por:

```

TVetor(const TVetor& obj)
{
//terão a mesma dimensão
this->dimensao = obj.dimensao;
this->v = NULL;
//Cria um novo vetor para v2

```

```

this->v = new int (n);
if(this->v == NULL)
    {cerr<<'\nFalha alocação'<<endl; exit(0);}
//copia os valores
for(int i = 0; i < dimensao; i++)
    this->v[i] = obj.v[i];
}

```

Observe que o novo objeto vai ter um vetor com a mesma dimensão e os mesmos valores.

No exemplo abaixo usa um construtor sobrecarregado, que recebe uma lista de parâmetros. Observe que os objetos são criados um a um dentro do for.

Exemplo:

```

//Criando 50 objetos dinâmicos
vector< TNomeClasse * > ptrObj(50);
for (int i = 0; i < 50; i++)
    ptrObj[i] = new TNomeClasse (parametros);

```

13.5 Métodos destrutores

Os métodos destrutores tem o objetivo de finalizar o objeto e liberar a memória alocada. Tem o mesmo nome da classe antecedido pelo til (~). São automaticamente executados quando o objeto sai de escopo.

Veja no exemplo da classe TVetor a forma do destrutor.

Exemplo:

```

//Destrutor da classe TVetor
~TVetor()
{
    delete v;
    v = NULL;
};
};

```

13.5.1 Sentenças para destrutores

- Em cada classe um único destrutor.
- Os destrutores são sempre públicos.
- Não retornam nada nem mesmo void.
- Não podem ter argumentos.
- Não podem ser sobrecarregados.
- Não podem ser const nem voláteis.

- Devem liberar a memória alocada no construtor e destruir os objetos dinâmicos.
- O corpo de um destrutor é executado antes dos destrutores para os objetos membros. De uma maneira geral, a ordem de destruição dos objetos é inversa a ordem de construção. Veremos posteriormente o uso de herança e de destrutores virtuais.
- ²Como regra básica, sempre declare o destrutor como virtual.
- ²Se durante a execução do programa é chamada a função `exit()` os construtores locais não são executados, os globais sim. Se for chamada a função `abort()` nenhum destrutor é executado.

13.5.2 Ordem de criação e destruição dos objetos

Quando um objeto é criado, são criados os objetos internos e depois é executado o método construtor. A seqüência de destruição é inversa a da criação, isto é, primeiro é executado o método destrutor e depois são eliminados os objetos internos da classe.

Se for criada uma matriz de objetos a seqüência de criação é `obj[0], obj[1], obj[2], ..., obj[n]` e a seqüência de destruição `obj[n], obj[n-1], obj[n-2], ..., obj[1], obj[0]`.

```
Exemplo:
//Arquivo X.h
class X
{
  int x;
  int * ptr;
  X():x(0)
    { ptr = new int (30);
    };
  ~X()
    {delete [] ptr;
    };
};
//Arquivo prog.cpp
#include "X.h"
void main()
{
  X objeto; //cria objeto
}; //objeto sai de escopo e é destruído
```

No exemplo acima, dentro da função `main` cria um objeto do tipo `X`. Primeiro cria o atributo `x` e inicializa com 0, a seguir cria `ptr`. Depois é executado o construtor da classe que aloca memória para um vetor de 30 inteiros e armazena o endereço do bloco alocado em `ptr`. Na seqüência de destruição, primeiro é executado o destrutor, que deleta o conteúdo do ponteiro (`delete ptr`), depois é eliminado o atributo `ptr` e depois o atributo `x`. Observe que a ordem de criação é inversa a de construção.

A melhor maneira de entender a seqüência de criação e destruição dos objetos é acompanhar a execução do programa em um debugger¹.

¹No Borland C++ 5.0, usa-se a função `f7`. Veja no capítulo 48 como usar o debugger da GNU.

13.6 Sentenças para construtores e destrutores

- Um objeto automático é construído na sua definição e destruído quando sai de escopo.

Exemplo:

```
{int x;      //objeto x é construído
}           //objeto x sai de escopo e é destruído
```

- Um objeto temporário, criado na avaliação de uma expressão é destruído no fim do bloco (ou do método). Ou seja, não armazene nenhuma informação (ponteiro, referência) em objetos temporários.
- Um objeto criado com `new` deve ser deletado com `delete`.
- Não destruir um objeto alocado dinamicamente não causa bug, mas é um desperdício de memória.
- Lembre-se, o operador `delete` primeiro chama o método destrutor do objeto, a seguir devolve a memória para o sistema operacional.
- ² O uso de uma lista de inicialização não é permitido se você criou um construtor com parâmetros.

Exemplo:

```
tipo a = {v1,v2,...vn};
```

- ³ Objetos globais estáticos são criados quando o programa é carregado (antes de `main`) e destruídos quando o programa é encerrado. Você pode criar uma função global estática e executá-la antes de `main` para iniciar um objeto estático.

Capítulo 14

Herança

O idéia da herança foi apresentada na seção 2.7, neste momento você poderia rever aquela seção. Neste capítulo vamos apresentar o uso do conceito de herança em C++, isto é, como implementar o conceito de herança usando C++.

O conceito de herança permite a criação de uma classe derivada, ou seja, dada uma classe base, pode-se criar uma classe derivada que herda todos os atributos e métodos da classe base.

14.1 Protótipo para herança

Veja o protótipo para herança:

Protótipo:

```
//-----arquivo Base.h
class Base
{
//Definição dos atributos e métodos da classe Base
};
//-----arquivo Derivada.h
#include "Base.h"
class Derivada: public Base //especificador de herança, seção 14.2
{
//Definição dos atributos e métodos da classe Derivada
};
```

Observe a linha,

```
class Derivada: public Base
```

nesta linha, informa-se que a classe Derivada é herdeira da classe Base.

O especificar public, é o especificador de herança e define a forma como a classe Derivada pode acessar os atributos e métodos da classe Base. O uso do especificador de herança é descrito na seção 14.2.

Veja na Figura 14.1 a classe TCirculo, herdeira de TPonto, que foi anteriormente apresentada.

Apresenta-se a seguir a classe TCirculo. A classe TCirculo é herdeira da classe TPonto apresentada anteriormente.

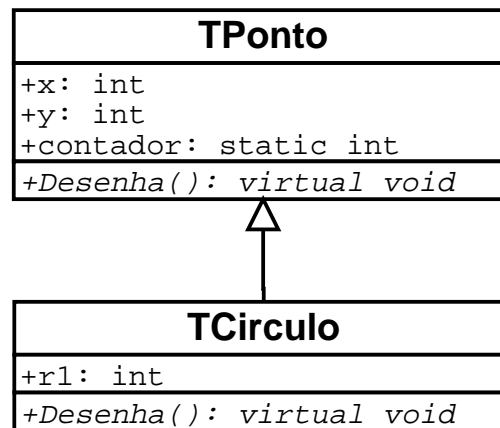


Figura 14.1: A herança entre TPonto e TCirculo.

Listing 14.1: Arquivo e87-TCirculo.h.

```

//-----Arquivo e87-TCirculo.h
#ifndef _TCirculo_
#define _TCirculo_

#include "e87-TPonto.h"

/*
Define o tipo de usuário TCirculo.
*/

class TCirculo : /*virtual*/ public TPonto
{
public:
int r1;

//Construtor
//observe que chama construtor da classe base
TCirculo(int _x,int _y, int _raio):TPonto(_x,_y),r1(_raio)
{
};

//sobrecarga
inline void Set(int x,int y, int raio);

//sobrecarga
inline void Set(TPonto & p, int raio);

//sobrecarga
inline void Set(TCirculo & );

//nova
int Getr1()const {return r1;};

//redefinida
virtual void Desenha();
};
  
```

```
#endif
```

Observe a herança com a classe TPonto. A classe TCirculo cria o atributo r1 e inclui o método Getr1(). Alguns métodos Set de TPonto são sobrecarregados (mesmo nome mas parâmetros diferentes). TCirculo redefine o método Desenha. Agora preste atenção no construtor da classe TCirculo, e observe a chamada do construtor da classe TPonto.

Listing 14.2: Arquivo e87-TCirculo.cpp.

```
//-----Arquivo e87-TCirculo.cpp
#include "e87-TCirculo.h"

#include <iostream>

//Implementação dos métodos de TCirculo
void TCirculo::Set(int x,int y, int raio)
{
  TPonto::Set(x,y);
  this->r1 = raio;
}

void TCirculo::Set(TPonto& p, int raio)
{
  Set(p.x,p.y);
  r1 = raio;
}

void TCirculo::Set(TCirculo & c)
{
  this->x = c.x;
  this->y = c.y;
  this->r1 = c.r1;
}

//Implementação de Desenha
//Usa o método desenha da classe pai e
//acrescenta o desenho do circulo
void TCirculo::Desenha()
{
  //chama função da classe base
  TPonto::Desenha();

  //instrução para desenhar o circulo;
  std::cout << "\nTCirculo:□Coordenada□r1=" << r1 <<std::endl;
}
```

14.2 Especificador de herança

O especificador de herança altera a forma como se processa a herança. Pode-se usar os especificadores public, protect e private.

Protótipo:

```
class Derivada: public Base {};
```

```
class Derivada: protected Base {};  
class Derivada: private Base {};
```

O acesso aos membros da classe Base vai depender do especificador de herança (public, protect e private) e dos especificadores de acesso na classe pai (public, protect e private)¹.

A Tabela 14.1 mostra o acesso herdado. Na primeira coluna o especificador de acesso utilizado na classe base, na segunda coluna o especificador de herança e na terceira coluna o acesso efetivamente herdado.

Observe na primeira linha da tabela, que se o atributo é public e o especificador de herança é public, o acesso herdado é public. Se o atributo é protected e o especificador de herança é private, o acesso herdado é private.

Tabela 14.1: Acesso herdado.

Tipo de acesso na classe base	Especificador de herança	Acesso herdado
public	public	public
protected	public	protected
private	public	inacessível
public	protected	protected
protected	protected	protected
private	protected	inacessível
public	private	private
protected	private	private
private	private	inacessível

Vamos esclarecer o uso do especificador de acesso em heranças através de um exemplo.

```
Exemplo:  
class A  
{  
public:  
    int x;  
protected:  
    int y;  
private:  
    int z;  
};  
class B: public A  
{  
int X() {return x;}; //ok x é público  
int Y() {return y;}; //ok y é protegido  
int Z() {return z;}; //Erro não tem acesso a z
```

¹Vimos na seção 8.2 o uso dos especificadores public, protected e private para definir o acesso aos atributos da classe. Lembre-se que usamos public para informar que o atributo pode ser acessado externamente.

```
};
class C: private A
{
int X() {return x;}; //ok x é privado
int Y() {return y;}; //ok y é privado
int Z() {return z;}; //Erro não tem acesso a z
};
```

Se um atributo `z` é `private` você sabe que ele só pode ser utilizado na classe onde foi declarada. Assim, se ocorrer um bug com o atributo `z`, você só precisa conferir o código da classe onde `z` foi definido.

Se um atributo é protegido, ele só pode ser utilizado na classe onde foi declarado e nas classes herdeiras, assim, ao procurar um bug, você só precisa procurar o bug na classe onde o atributo foi declarado e nas classes herdeiras.

Se um atributo público está causando um bug, você terá de procurar o mesmo em toda a hierarquia de classes e no restante do seu programa, isto é, em todos os locais onde a classe é utilizada.

Observe que quando mais encapsulado o seu código mais fácil encontrar os erros e bugs.

14.3 Chamando construtores da classe base explicitamente

Se a classe base `A` só tem construtores com parâmetros (não tem o construtor default), a classe derivada `B` deve chamar explicitamente um dos construtores de `A`.

```
Exemplo:
//-----A.h
class A {
int p1,p2;
//declaração do construtor (não tem construtor default)
A(int p1,int p2);
};
//-----A.cpp
//Definição do construtor
A::A(_p1,_p2): p1(_p1),p2(_p2){};
//-----B.h
class B: public A
{
int p3;
B(int p1,int p2,int p3);
};
//-----B.cpp
//Observe abaixo a chamada do construtor de A
//passando os parâmetros _p1 e _p2
B::B(int _p1,int _p2,int _p3):A(_p1,_p2),p3(_p3)
{};
```

14.4 Ambigüidade

Uma ambigüidade ocorre quando o compilador não consegue identificar qual atributo ou método deve ser acessado. O exemplo abaixo mostra uma ambigüidade.

Exemplo:

Aló, com quem deseja falar?

Com o Fernando.

Mas qual Fernando, o Henrique ou o Collor ?.

De uma maneira geral vai ocorrer uma ambigüidade quando no mesmo nível tivermos atributos ou métodos com o mesmo nome, ou seja, se um nome domina o outro não vai existir ambigüidade.

A verificação de ambigüidade é feita antes do controle de acesso, assim, primeiro o compilador verifica se o objeto não é ambiguo e depois se o mesmo pode ser acessado.

As ambigüidades podem ser resolvidas explicitamente com o operador de resolução de escopo (::).

Ao tentar compilar o exemplo abaixo aparece uma mensagem de erro. O construtor default esta definido duas vezes.

Listing 14.3: Erro ao definir duas vezes o construtor default (Arquivo e101-ambigüidade.cpp).

```
//Cuidado com Inicializadores
class X
{
int x;

public:
//Sejam os construtores
//construtor1
X(){};

//construtor2
X(int _x = 0)
{x = _x;};
};

void main()
{
//Se na tentativa de criar um objeto você faz:
X obj1(5); //Cria objeto 1, usa construtor2

//A linha abaixo tem uma ambigüidade
X obj2;    //Qual construtor? X() ou X(int i=0)
};

/*
Saída gerada pelo compilador:
-----
[andre@mercurio Cap3-P00UsandoC++]$ g++ e101-ambigüidade.cpp
e101-ambigüidade.cpp: In function 'int main (...)':
e101-ambigüidade.cpp:22: call of overloaded 'X()' is ambiguous
e101-ambigüidade.cpp:9: candidates are: X::X ()
e101-ambigüidade.cpp:13: X::X (int = 0)
```



```
*/  
/*  
Observe a mensagem  
e101-ambiguidade.cpp:22: call of overloaded 'X()' is ambiguous  
  
o compilador não consegue identificar qual construtor deve ser executado.  
*/
```

14.4.1 Sentenças para herança

- A classe base de uma hierarquia é chamada de classe base, de classe pai ou de superclasse. As demais classes são denominadas classes derivadas, subclasses ou classes filhas.
- A maior dificuldade em uma hierarquia é modelar com clareza as classes e seus relacionamentos.
- O uso de `protected` e `private` deixa o código mais encapsulado. Um código mais encapsulado é mais facilmente depurado.
- Observe que a classe derivada não altera em nada a classe base, de forma que se a classe derivada for alterada, somente ela precisa ser recompilada.
- Uma classe vazia pode ser utilizada temporariamente quando o programador quer criar uma classe base mas ainda não identificou as relações entre as classes derivadas. Uma classe vazia pode criar objetos.
- A declaração de um atributo `x` numa classe derivada, que já tenha sido definido na classe base oculta o atributo da classe base. Para acessar o atributo da classe base use o operador de resolução de escopo (`::`).

Exemplo:

```
Base::x;
```

- Numa herança é aconselhável que um método construtor chame os métodos construtores ancestrais para a declaração dos atributos ancestrais, em vez de declarar ele mesmo estes atributos.
- ²O operador de atribuição e os construtores da classe base não são herdados pelas classes derivadas, ou seja, todas as classes devem ter construtores, destrutores e operadores sobrecarregados.
- ²O uso de classes, herança e métodos virtuais é a base para o polimorfismo, que possibilita o desenvolvimento de sistemas complexos.
- ² Se você tem uma herança, os destrutores devem ser virtuais. Se você tiver herança e destrutores não virtuais, na destruição de um objeto dinâmico, só será executado o destrutor de mais alto nível (o da classe base).

Capítulo 15

Herança Múltipla²

Apresenta-se neste capítulo o protótipo e o conceito de herança múltipla. Os problemas de ambiguidade em herança múltipla. A seguir apresenta-se a herança múltipla virtual e a ordem de criação e destruição dos objetos em uma herança.

15.1 Protótipo para herança múltipla

Veja a seguir o protótipo para os diferentes tipos de herança múltipla.

Protótipos:

```
//Herança múltipla, seção 15.2
class Derivada : especificador_herança Base1, especificador_herança Base2
{};
//Herança múltipla virtual, seção 15.4
class Derivada : virtual especificador_herança Base1, virtual especificador_herança Base2
{};
```

15.2 Herança múltipla

A herança múltipla ocorre quando uma classe derivada é herdeira de mais de uma classe base.

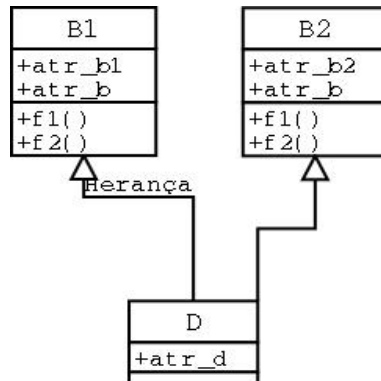
Veja no exemplo da Figura 15.1, que a classe D é herdeira das classes B1 e B2.

Veja no exemplo a seguir como implementar a herança múltipla apresentada na Figura 15.1. A herança se dá da esquerda para a direita, primeiro herda os atributos e métodos de B1, depois de B2.

Observe na Figura 15.2 como ficam os objetos b1, b2 e d na memória de seu computador. O objeto b1 tem o atributo atr_b1, o objeto b2 tem o atributo atr_b2 e o objeto d os atributos atr_b1 (herdado de B1), atr_b2 (herdado de B2) e atr_d (da própria classe D).

```
Exemplo:
class B1
    {int atr_b1;};
class B2
    {int atr_b2;};
class D: public B1, public B2
```

Figura 15.1: Herança múltipla.



```

    {int atr_d;};
int main()
{
  B1 b1;
  B2 b2;
  D d;
}

```

Figura 15.2: Como ficam os objetos b1, b2 e d em uma herança múltipla.

b1	b2	d
atr_b1	atr_b2	atr_b1
		atr_b2
		atr_d

15.3 Ambiguidade em herança múltipla

Quando você usa herança múltipla pode ocorrer que as classes B1 e B2 tem um atributo com o mesmo nome (veja na Figura 15.1 o atributo `atr_b`), este atributo vai ser criado duas vezes uma pelo caminho D-B1 e outra pelo caminho D-B2. Quando você tentar acessar este atributo, o compilador exibirá uma mensagem de erro por ambigüidade, ou seja, quando você acessa `atr_b` você quer acessar `atr_b` da classe B1 ou `atr_b` da classe B2 ?

Para contornar este problema, você deve usar o operador de resolução de escopo (`::`), veja o protótipo abaixo.

Protótipo:

```

base1::nomeAtributoAmbiguo;
base2::nomeAtributoAmbiguo;

```

```
Exemplo:
B1::atr_b;
B2::atr_b;
```

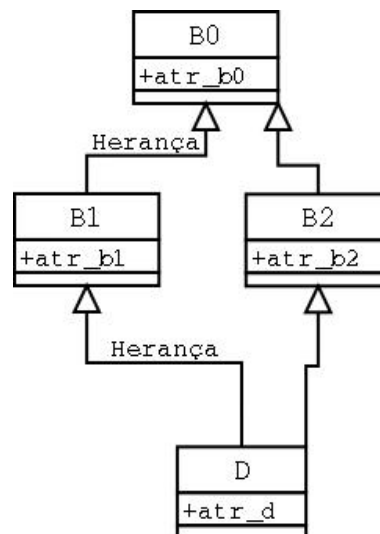
Pode-se eliminar o problema da ambiguidade com uso da palavra chave `using`¹. Com as declarações `using` você pode selecionar, numa classe derivada, os métodos e atributos a serem utilizados.

```
Exemplo:
class D: public B1, public B2
{
using B1::f1;    //quando chamar f1 usar f1 da classe B1
using B2::f2;    //quando chamar f2 usar f2 da classe B2
using B1::atr_b; //quando chamar atr_b usar atr_b da classe B1
};
```

15.3.1 Herança múltipla com base comum

O diagrama ilustrado na Figura 15.3, mostra uma herança múltipla com base comum. A classe D é herdeira das classes B1 e B2 e estas são herdeiras de B0, ou seja, a classe B0 é uma classe base comum.

Figura 15.3: Herança múltipla com base comum.



Veja a seguir como implementar a hierarquia da Figura 15.3.

```
Exemplo:
class B0
{int atr_b0;};
class B1 : public B0
```

¹Veja descrição detalhada dos métodos de resolução de escopo com `using` na seção 7.1.

```

    {int atr_b1;};
class B2 : public B0
    {int atr_b2;};
class D: public B1, public B2
    {int atr_d;};
int main()
{
B1 b1;
B2 b2;
D d;
}

```

Quando você tem herança múltipla com D herdando de B1 e B2, a classe D vai percorrer os construtores das classes B1 e B0 e a seguir os construtores de B2 e B0, criando duas vezes o atributo `atr_b0`. Observe na Figura 15.4 que o atributo `atr_b0` é criado duas vezes, uma pelo caminho `D::B1::B0` e outra pelo caminho `D::B2::B0`.

Figura 15.4: Como ficam os objetos `b1`, `b2` e `d` em uma herança múltipla com base comum.

b1	b2	d
atr_b0	atr_b0	B1::atr_b0
		B2::atr_b0
atr_b1	atr_b2	atr_b1
		atr_b2
		atr_d

15.4 Herança múltipla virtual²

Dica: Este é um título de nível 2, ou seja, se estiver lendo a apostila pela primeira vez, pule esta parte.

Vimos na seção 15.3 que ao criar uma classe derivada de mais de uma classe base (herança múltipla), o compilador cria todos os atributos da classe base1 e depois todos os atributos da classe base2. Vimos ainda na seção 15.3.1 que se as classes base1 e base2 são herdeiros da classe base0, então o compilador vai criar os atributos de base1 e base0 e a seguir de base2 e base0, de forma que o atributo de base0 (`atr_b0`) será criado 2 vezes.

Se você deseja ter o atributo repetido, tudo bem. Porém, se você deseja ter apenas uma cópia do mesmo, você precisa definir a herança como virtual. O protótipo abaixo mostra a sintaxe da herança virtual.

Protótipo:

```

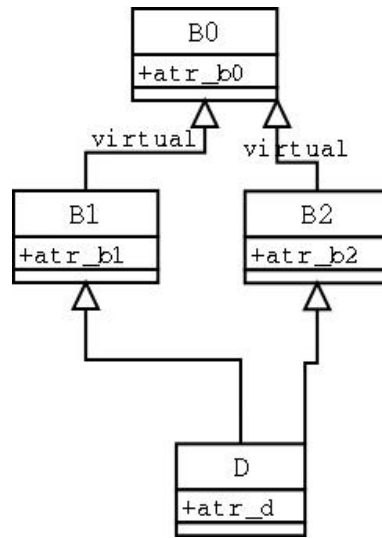
class nomeClasseDerivada: public virtual base1, public virtual base2
{};

```

Veja na Figura 15.5 uma herança múltipla virtual. Neste caso, como a herança é declarada como virtual, só será criada uma cópia do atributo da classe base, ou seja, com a especificação virtual, `atr_b0` só é criado uma vez, pelo caminho D-B1-B0.

Veja no exemplo a seguir como implementar a hierarquia da Figura 15.5, observe o uso da palavra chave virtual. Observe na Figura 15.6 que o atributo `atr_b0` é criado apenas uma vez, pelo caminho `D::B1::B0`.

Figura 15.5: Herança múltipla virtual.



Exemplo:

```

class B0
{int atr_b0;};
class B1: virtual public B0
{int atr_b1;};
class B2: virtual public B0
{int atr_b2;};
class D: virtual public B1, virtual public B2
{int atr_d1;};
  
```

Figura 15.6: Como ficam os objetos `b1`, `b2` e `d` em uma herança múltipla com base `B0` comum e virtual.

b1	b2	d
atr_b0	atr_b0	atr_b0
atr_b1	atr_b2	atr_b1
		atr_b2
		atr_d

15.4.1 Sentenças para herança múltipla

- Uma classe D não pode ter derivação múltipla de uma classe B, este procedimento não é permitido por ser ambíguo.

Exemplo:

```
class B{};
class D:public B, public B{}; //Erro ambíguo
```

- Você pode criar um método de inicialização (InicializaAtributos()), que pode ser usado para inicialização dos atributos do objeto. Você pode chamar InicializaAtributos no construtor da classe derivada.
- Observe que a chamada do construtor da classe base só vai ser importante se ele for efetivamente implementado e contiver por exemplo a criação de objetos dinâmicos.
- O uso de herança virtual é interessante, permitindo construir classes mais complexas a partir da montagem de classes mais simples, com economia de memória e de implementação.
- A herança virtual deve ser usada com critério e com cuidado.
- ²Se a classe base é declarada como virtual, ela tem de ter um construtor default. Isto é necessário para a criação de uma matriz de objetos, pois quando se cria uma matriz de objetos, o construtor executado é o default.
- ²Se numa herança virtual a classe base não tiver um construtor default (pois foi definido um construtor com parâmetros). O construtor da classe base virtual deve ser explicitamente chamado.

Exemplo:

```
Class TMinhaJanela: public TFrameWindow
{
//construtor
TMinhaJanela(string titulo);
};
//Definição do construtor
TMinhaJanela::TMinhaJanela(string titulo)
: TframeWindow(titulo) //Chama classe base
, Twindow(titulo) //Chama classe base virtual explicitamente
{.}
```

- ²Se você deseja ter um mecanismo para identificar de que classe é um objeto (dado que você tem apenas um ponteiro para a classe base), você deve incluir um método que retorne o nome da classe. Isto é utilizado na programação em ambientes de janela (como Windows, Max OS-X, Gnome/KDE), em que cada classe tem um método que retorna o nome da classe, este método é chamado GetClassName(). Mais recentemente o C++ incluiu typeid que retorna dinamicamente o tipo de um objeto.

- ²Quando se usa o mecanismo do polimorfismo é fundamental que o destrutor da classe base seja virtual, se o destrutor da classe base não for virtual o destrutor da classe derivada não vai ser executado.
- ²Procure criar uma classe base que tenha somente métodos virtuais.

15.5 Ordem de criação e destruição dos objetos em heranças²

Numa herança múltipla, a sequência de criação dos objetos se da esquerda para a direita, a Figura 15.7 ilustra, na primeira coluna, a sequência de criação dos objetos. Primeiro executa o construtor de A, depois o construtor de AA e finalmente o construtor de AAA. A sequência de destruição, ilustrada na segunda coluna, é inversa a da criação, primeiro executa o destrutor de AAA depois o destrutor de AA e finalmente o destrutor de A.

Figura 15.7: Seqüência de construção e destruição dos objetos em uma herança.

//Criação	//Destruição
construtor de A();	destrutor de ~AAA();
construtor de AA();	destrutor de ~AA();
construtor de AAA();	destrutor de ~A();

15.5.1 Ordem de criação e destruição dos objetos em heranças virtuais

No exemplo a seguir uma classe X tem herança normal de Y e virtual de Z.

```
Exemplo:
//classe X com herança virtual de Z
class X: public Y, virtual public Z
{
};
//Cria objeto do tipo X com nome obj
X obj;
```

A sequência de criação de obj é dada por Z(), Y() , X().

Veja na Figura 15.8 uma hierarquia de classes com heranças normais e virtuais (B1, B2, D1, D2 e Top). As classes base são B1 e B2. A classe D1 tem derivação normal de B2 e virtual de B1. A classe D2 tem derivação normal de B2 e virtual de B1. A classe Top tem derivação normal de D1 e virtual de D2.

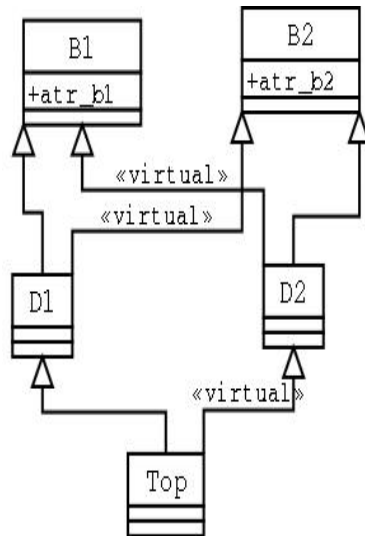
A hierarquia da Figura 15.8 é implementada da forma:

Listing 15.1: Seqüência de construção e destruição em herança múltipla virtual.

```
#include <iostream>
using std::cout;
```

²Novamente, devo lhe lembrar que o melhor método para aprender a ordem de criação e destruição em hierarquias é usar um debug.

Figura 15.8: Hierarquia com herança múltipla normal e virtual.



```

using std::endl;

class B1
{
int atr_b1;
public:
    B1(){cout<<"Construtor_B1"<<endl;};
    ~B1(){cout<<"Destrutor_B1"<<endl;};
};

class B2
{
int atr_b2;
public:
    B2(){cout<<"Construtor_B2"<<endl;};
    ~B2(){cout<<"Destrutor_B2"<<endl;};
};

class D1: public B2, virtual public B1
{
public:
    D1(){cout<<"Construtor_D1"<<endl;};
    ~D1(){cout<<"Destrutor_D1"<<endl;};
};

class D2: public B2, virtual public B1
{
public:
    D2(){cout<<"Construtor_D2"<<endl;};
    ~D2(){cout<<"Destrutor_D2"<<endl;};
};

class Top: public D1, virtual public D2
  
```

```

{
public:
    Top() {cout<<"Construtor_Top"<<endl;};
    ~Top() {cout<<"Destrutor_Top"<<endl;};
};

void main()
{
    Top p;
}

/*
Saída:
-----
[andre@mercurio Cap3-P00UsandoC++]$ ./a.out
Construtor B1
Construtor B2
Construtor D2
Construtor B2
Construtor D1
Construtor Top
Destrutor Top
Destrutor D1
Destrutor B2
Destrutor D2
Destrutor B2
Destrutor B1
*/

```

A sequência de criação do objeto p é dada por:

- Primeiro a sequência de D2 que é virtual: cria B1 que é virtual, cria B2 e então cria D2.
- Depois a sequência de D1 que é normal: B1 é virtual pelo caminho Top::D2::B1 e já foi criado, cria B2 e depois cria D1.
- Finalmente cria Top.

Ou seja, os construtores default das classes virtuais são chamados antes dos das classes não virtuais.

³Releia a frase acima, veja que escrevi *os construtores default das classes virtuais*. Se a classe base virtual não tem um construtor default, um dos construtores com parâmetros deve ser explicitamente chamado. Lembre-se, toda classe tem um construtor default, sem parâmetros, que é criado automaticamente pelo compilador. Vimos que você pode reescrever o construtor da forma que quiser, e que se você criar qualquer construtor com parâmetros, o construtor default deixa de ser criado pelo compilador. Como o construtor default deixa de ser criado, o construtor existente precisa ser explicitamente chamado.

15.6 Redecaração de método ou atributo na classe derivada

Na Parte I da apostila, falamos que uma classe derivada herda os atributos e os métodos da classe base, e que a forma como as coisas acontecem na classe derivada são um pouco diferentes da classe

base. Mas para que os objetos derivados sejam efetivamente diferentes, você precisa redeclarar e redefinir alguns métodos da classe base.

A classe derivada pode acrescentar novos métodos, ou alterar os métodos da classe base. Podemos ter dois casos: No primeiro, a classe derivada realiza a mesma operação que a classe base, mas de uma outra forma. No segundo, a classe derivada realiza a mesma operação da classe base e mais alguma coisa. Isto significa que a classe derivada pode reescrever totalmente o método herdado, ou pode chamar o método da classe base e depois acrescentar alguma coisa.

15.6.1 Sentenças para redeclarações

- Se você criar um atributo na classe derivada com o mesmo nome da classe base, o atributo da classe base continua existindo, mas para acessá-lo você deve usar o operador de resolução de escopo (::).

– Exemplo:

– `NomeClasseBase::nomeAtributoAmbíguo;`

- Se você redeclarar um método na classe derivada, este oculta o método de mesmo nome da classe base. O método da classe base pode ser acessado utilizando-se o operador de resolução de escopo.

Exemplo:

`NomeClasseBase::nomeMétodoAmbíguo();`

- Quando você inibe um método da classe base, redeclarando o mesmo, o faz para implementar o mesmo de uma forma diferente. Normalmente o método da classe derivada faz o que a classe base fazia de uma maneira diferente, ou faz o que a classe base fazia e mais alguma coisa.
- ²Não confundir redeclaração (método com mesmo nome e parâmetros na classe derivada), com uso de métodos virtuais (método com mesmo nome e parâmetros mas com especificador virtual) e com uso de sobrecarga de métodos (métodos com mesmo nome mas com parâmetros diferentes).

15.7 Exemplo de herança simples e herança múltipla

Observe na Figura 15.9 a hierarquia de classes TPonto. A classe TCirculo é herdeira de TPonto e a classe TElipse é herdeira de TCirculo. A classe TCirculoElipse é herdeira de TCirculo e de TElipse.

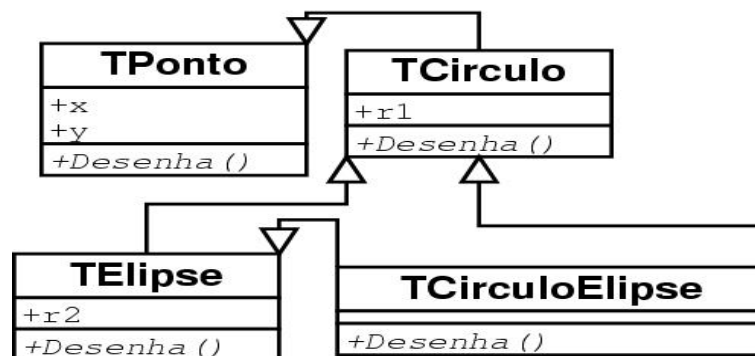
Veja nas listagens a seguir o uso de herança simples e herança múltipla.

As classes TPonto e TCirculo foram anteriormente apresentadas. Apresenta-se aqui a listagem das classes TElipse, TCirculoElipse e um programa que usa as classes definidas.

Listing 15.2: Arquivo e87-TElipse.h.

```
//-----Arquivo e87-TElipse.h
#ifndef _TElipse_
#define _TElipse_
```

Figura 15.9: Hierarquia de classes TPonto.



```

#include "e87-TCirculo.h"

//Herança simples
class TElipse:public TCirculo
{
public:
    int r2;

//construtor
TElipse (int x,int y, int raiol ,int raio2);

//set
void Set (int x,int y, int raiol ,int raio2);

//redefinida
virtual void Desenha ();
};

#endif

```

Listing 15.3: Arquivo e87-TElipse.cpp.

```

//-----Arquivo e87-TElipse.cpp
#include "e87-TElipse.h"

#include <iostream>

//Construtor de TElipse,
//observe que a chamada explicita do construtor da classe
//base TCirculo é necessário porque TCirculo não tem um
//construtor default e quero passar os atributos x,y e raio1
TElipse::TElipse (int x,int y, int raiol ,int raio2)
    :TCirculo(x,y,raio1), r2(raio2)
{
}

void TElipse::Set (int x,int y, int raiol ,int raio2)
{

```

```

    //herdou x e y de TPonto
this->x = x;
this->y = y;

    //herdou r1 de TCirculo
r1 = raio1;

    //criou o atributo r2 na classe TElipse
r2 = raio2;
}

void TElipse::Desenha()
{
    //Instrução para desenhar o circulo;
TCirculo::Desenha();

    //Acrescenta coisas novas,
std::cout << "\nTElipse:□Coordenada□r2=" << r2 <<std::endl;
}

/*
Observação:
Observe que Desenha de TElipse chama Desenha de TCirculo
e depois acrescenta coisas novas.
Isto é, o método Desenha da classe base é redefinido,
fazendo o que TCirculo::Desenha fazia e mais algumas coisa.
*/

```

Listagem do programa e87-Programa.cpp.

Listing 15.4: Arquivo e87-Programa.cpp.

```

//-----Arquivo e87-Programa.cpp
#include <iostream>
using namespace std;

#include "e87-TPonto.h"
#include "e87-TCirculo.h"
#include "e87-TElipse.h"

//Exemplo de criação e uso dos objetos TPonto, TCirculo e TElipse
int main()
{
    //Parte I teste de TPonto
    int x = 5;
    int y = 4;
    {
        cout << "\n-----Testando□TPonto:"<<endl;

        //Cria objeto do tipo TPonto com nome ponto
        TPonto ponto;

        //Chama método Set do objeto ponto
        ponto.Set(x,y);

        //Chama método Desenha do objeto ponto

```

```

    ponto.Desenha();
} //sai de escopo e detroe ponto

    cout << "\n-----Testando TPonto dinâmico:"<<endl;

//Cria ponteiro para TPonto
TPonto* ptr = NULL;

//Cria objeto do tipo TPonto e coloca endereço em ptr
ptr = new TPonto;

//Chama método Set do objeto ptr passando os valores de x e y
x = 6; y=7;
ptr->Set(x,y);
ptr->Desenha();
int xx = ptr->Getx();

//chama método estático da classe TPonto
//observe que não precisa de um objeto
//usa o nome da classe seguido do operador de resolução de escopo.
cout<< "Contador="<<TPonto::GetContador()<<endl;
delete ptr;

//Parte II teste de TCirculo
    cout << "\n-----Testando TCirculo:"<<endl;
    TCirculo c (55,44,33);
    c.Desenha();

//Parte III teste de TElipse
    cout << "\n-----Testando TElipse:"<<endl;
    TElipse e (555,444,333,222);
    e.Desenha();
}

/*
Para compilar no Linux:
g++ e87-Programa.cpp e87-TCirculo.cpp e87-TElipse.cpp e87-TPonto.cpp
*/
/*
Saída:
-----
[andre@mercurio Cap3-P00UsandoC++]$ ./a.out

-----Testando TPonto:

TPonto: Coordenada x=5
TPonto: Coordenada y=4

-----Testando TPonto dinâmico:

TPonto: Coordenada x=6
TPonto: Coordenada y=7
Contador = 1

-----Testando TCirculo:

```

```

TPonto: Coordenada x=55
TPonto: Coordenada y=44
TCirculo: Coordenada r1=33

-----Testando TElipse:
TPonto: Coordenada x=555
TPonto: Coordenada y=444
TCirculo: Coordenada r1=333
TElipse: Coordenada r2=222
*/

```

No exemplo a seguir define-se a classe TCirculoElipse que representa um olho. A classe TCirculoElipse apresenta erros que serão discutidos a seguir.

Listing 15.5: Arquivo e87-TCirculoElipse.h.

```

//-----Arquivo e87-TCirculoElipse.h
#ifndef _TCirculoElipse_
#define _TCirculoElipse_

#include <iostream>
using namespace std;

#include "e87-TCirculo.h"
#include "e87-TElipse.h"

//Quero um circulo e uma elipse (um olho),
//as coordenadas do ponto central são as mesmas.
//Herança múltipla, herda de TCirculo e de TElipse

class TCirculoElipse: public TCirculo, public TElipse
{
public:

//construtor
TCirculoElipse (int xc,int yc, int rc,int r1e,int r2e);

//construtor de conversão
TCirculoElipse(TCirculo& circulo);

inline void Set (int xc,int yc, int rc,int r1e,int r2e);

//redefinida
virtual void Desenha ();
};
#endif

```

Listing 15.6: Arquivo e87-TCirculoElipse.cpp.

```

//-----Arquivo e87-TCirculoElipse.cpp
#include "e87-TCirculoElipse.h"

//Construtor
TCirculoElipse::TCirculoElipse(int xc,int yc, int rc,int r1e, int r2e)

```



```

{
//uso do operador de resolução de escopo para acessar método ancestral
TCirculo::Set(xc,yc,rc);
TElipse::Set(xc,yc,r1e,r2e);
}

void TCirculoElipse::Set (int xc,int yc, int rc,int r1e, int r2e)
{
TCirculo::Set(xc,yc,rc);
TElipse::Set(xc,yc,r1e,r2e);
}

//Construtor de conversão
//Como o circulo não preenche totalmente o TCirculoElipse
//e quero construir um objeto do tipo TCirculoElipse a partir
//de um TCirculo, crio um construtor de conversão
TCirculoElipse (TCirculo& circulo)
{
TCirculo::Set(circulo);

//Observe abaixo que passa circulo.r1 como r1 e r2 da TElipse
TElipse::Set(circulo.x,circulo.y,circulo.r1,circulo.r1);
}

//Implementação de Desenha
//Abaixo o uso do operador de resolução de escopo para
//identificar o método da classe base
void TCirculoEPonto:: Desenha ()
{
//Desenha a elipse
TElipse::Desenha ();

//Desenha o circulo
TCirculo::Desenha ();
}

```

15.8 Análise dos erros emitidos pelo compilador²

A classe TCirculoElipse não compila devido a presença de alguns erros. O primeiro erro esta na ambiguidade no acesso a classe TCirculo (linhas 1-6) da listagem a seguir. Isto ocorre porque a classe TCirculo pode ser acessada pelo caminho TCirculo e pelo caminho TElipse::TCirculo. Outro erro apresentado é a falta da chamada explicita dos construtores de TCirculo, como TCirculo não tem um construtor default, o construtor de TCirculo precisa ser explicitamente chamado (linhas 7-11) o erro se repete para TElipse (linhas 12-15). Nas linhas 15-18, erro ao tentar converter TCirculoElipse para TCirculo, pois tem dois TCirculo e a base é ambígua. O Erro se repete nas linhas 21-24. Existe um erro na definição do método construtor TCirculoElipse (linhas 25-36), esta faltando o nome da classe e ::, isto é, TCirculoElipse::TCirculoElipse(...).

Listing 15.7: Exemplo de mensagem de erro emitida pelo compilador g++ (no Linux) - Arquivo e87-TCirculoElipse.msg.

```
1 [andre@mercurio Cap3-P00UsandoC++]$ g++ -c e87-TCirculoElipse.cpp
```

```

2 In file included from e87-TCirculoEllipse.cpp:2:
3 e87-TCirculoEllipse.h:30: warning: direct base 'TCirculo' is inaccessible
4 in 'TCirculoEllipse' due to ambiguity
5 e87-TCirculoEllipse.cpp: In method 'TCirculoEllipse::TCirculoEllipse (int,
6 int, int, int)':
7 e87-TCirculoEllipse.cpp:6: no matching function for call to
8 'TCirculo::TCirculo()'
9 e87-TCirculo.h:18: candidates are: TCirculo::TCirculo (int, int, int)
10 e87-TCirculo.h:35: TCirculo::TCirculo (const TCirculo
11 &)
12 e87-TCirculoEllipse.cpp:6: no matching function for call to
13 'TElipse::TElipse ()'
14 e87-TElipse.h:14: candidates are: TElipse::TElipse (int, int, int, int)
15 e87-TElipse.h:21: TElipse::TElipse (const TElipse&)
16 e87-TCirculoEllipse.cpp:8: cannot convert a pointer of type
17 'TCirculoEllipse' to a pointer of type 'TCirculo'
18 e87-TCirculoEllipse.cpp:8: because 'TCirculo' is an ambiguous base class
19 e87-TCirculoEllipse.cpp: In method 'void TCirculoEllipse::Set (int, int,
20 int, int, int)':
21 e87-TCirculoEllipse.cpp:14: cannot convert a pointer of type
22 'TCirculoEllipse' to a pointer of type 'TCirculo'
23 e87-TCirculoEllipse.cpp:14: because 'TCirculo' is an ambiguous base
24 class
25 e87-TCirculoEllipse.cpp: At top level:
26 e87-TCirculoEllipse.cpp:21: parse error before '&'
27 e87-TCirculoEllipse.cpp:24: 'circulo' was not declared in this scope
28 e87-TCirculoEllipse.cpp:24: 'circulo' was not declared in this scope
29 e87-TCirculoEllipse.cpp:24: 'circulo' was not declared in this scope
30 e87-TCirculoEllipse.cpp:24: 'circulo' was not declared in this scope
31 e87-TCirculoEllipse.cpp:24: ISO C++ forbids declaration of 'Set' with no
32 type
33 e87-TCirculoEllipse.cpp:24: 'int TElipse::Set' is not a static member of
34 'class TElipse'
35 e87-TCirculoEllipse.cpp:24: initializer list being treated as compound
36 expression
37 e87-TCirculoEllipse.cpp:25: parse error before '}'
38 e87-TCirculoEllipse.cpp:30: syntax error before '::'
39 e87-TCirculoEllipse.cpp:36: ISO C++ forbids declaration of 'Desenha'
40 with no type
41 e87-TCirculoEllipse.cpp:36: new declaration 'int TCirculo::Desenha()'
42 e87-TCirculo.h:34: ambiguates old declaration 'void TCirculo::Desenha
43 ()'
44 e87-TCirculoEllipse.cpp:36: declaration of 'int TCirculo::Desenha()'
45 outside of class is not definition
46 e87-TCirculoEllipse.cpp:37: parse error before '}'

```

Como você pode ver, a saída gerada pelo compilador pode ser muito grande. De um modo geral, um pequeno erro de digitação (de sintaxe) pode gerar várias mensagens de erro. Um exemplo clássico de erro de digitação e que gera uma saída confusa é a falta ou excesso de colchetes “{”,”}”.

Daí você conclue que é necessário compreender bem a sintaxe de C++ e digitar o programa com cuidado. Você precisa se concentrar e pensar no que está fazendo.

Capítulo 16

Polimorfismo²

Apresenta-se neste capítulo os métodos normais e os métodos virtuais que possibilitam a implementação do conceito de polimorfismo. Apresenta-se então como implementar o polimorfismo através de um exemplo. No final do capítulo apresenta-se os métodos virtuais puros.

16.1 Métodos não virtuais (normais, estáticos)

Os métodos de uma classe podem ser normais, estáticos ou virtuais.

Os métodos normais e os métodos estáticos tem seu endereço definido a priori, na compilação, tendo uma ligação estática. O termo ligação estática significa que quando o compilador encontra uma chamada do método, substitue a chamada pelo endereço do método. Quando o método é normal ou estático o endereço é fixo (estático).

Veja na Tabela 16.1 exemplos de métodos com ligação estática.

Tabela 16.1: Métodos com ligação estática.

	Préfixo	Retorno	Nome	Parâmetro	Sufixo
	<code>--</code>	Tipo	Nome	(Tipo p)	<code>--</code>
	<code>inline</code>	Tipo	Nome	(Tipo p)	<code>--</code>
	<code>static</code>	Tipo	Nome	(Tipo p)	<code>--</code>
	<code>inline static</code>	Tipo	Nome	(Tipo p)	<code>--</code>
	<code>--</code>	Tipo	Nome	(Tipo p)	<code>const</code>
	<code>inline</code>	Tipo	Nome	(Tipo p)	<code>const</code>
	<code>static</code>	Tipo	Nome	(Tipo p)	<code>const</code>
	<code>inline static</code>	Tipo	Nome	(Tipo p)	<code>const</code>

Isto esta um pouco confuso, vamos tentar esclarecer com um exemplo.

Exemplo:

```
/*Seja a classe A, com os métodos f1, f2, f3,  
verifique que f3 chama f1 e f2.*/  
class A
```

Tabela 16.2: Métodos com ligação dinâmica.

	Prefixo	Retorno	Nome	Parâmetro	Sufixo
	virtual	Tipo	Nome	(Tipo p)	__
	virtual	Tipo	Nome	(Tipo p)	const

```

{
void f1(){...};
void f2(){...};
void f3(){f1(); f2();};
};
//Seja a classe B, com redefinição de f1 e f2.
class B:public A
{
void f1(){...};
void f2(){...};
};
//Em main(), cria um objeto do tipo B com nome b
//e chama os métodos f1, f2, e f3 da classe B
void main()
{
B b;
b.f1(); //chama f1 da classe B
b.f2(); //chama f2 classe B
b.f3(); //chama f3 da classe A
}

```

O método f3 vai chamar os métodos f1 e f2 da classe A e não da classe B, ou seja, o compilador traduz main da seguinte forma.

```

void main()
{
B b;
B::f1(b); //chama f1 da classe B
B::f2(b); //chama f2 classe B
A::f3(b); //chama f3 da classe A
//o comando A::f3(b); vai ser traduzido por
//A::f1(b); A::f2(b);
}

```

Observe que foi criado um objeto do tipo B e que o método f3 esta executando f1 e f2 da classe A. Para que o método f3 chame os métodos f1 e f2 da classe B, f1 e f2 devem ser declaradas como métodos virtuais.

Veja a seguir como funcionam os métodos virtuais e como usar os métodos virtuais para que o método f3 do objeto B chame f1 e f2 do objeto B..

16.2 Métodos virtuais

Os métodos virtuais tem seu endereço definido dinamicamente, ou seja, durante a execução do programa. Nesta seção, vamos discutir como isto funciona.

Veja a seguir o protótipo de declaração de um método virtual. A única diferença em relação aos métodos normais é a inclusão da palavra chave `virtual`.

Protótipo:

```
class Nome
{
//Declara método virtual
virtual tipo funcao1();
//Declara método virtual com parâmetros
virtual tipo nome_função(parâmetros);
}
```

Veja na Tabela 16.2 exemplos de métodos com ligação dinâmica.

A classe que tem métodos virtuais, contém uma tabela chamada Virtual Memory Table (VMT, ou tabela de métodos virtuais). Esta tabela contém o endereço dos métodos virtuais. Assim, quando o programa esta sendo rodado, a definição de qual método deve ser acessado é dada naturalmente pela VMT.

Como funciona ?

A classe base tem o endereço de seus métodos virtuais na sua tabela VMT, e acessa os mesmos verificando seus endereços na VMT. A classe derivada também tem uma VMT, onde tem o endereço de seus métodos virtuais. Desta forma, quando um objeto da classe derivada chama um método, o endereço é obtido em tempo de execução da sua VMT.

Através deste mecanismo, a identificação de qual método vai ser executado é realizada dinamicamente através da verificação do endereço dos métodos na tabela VMT.

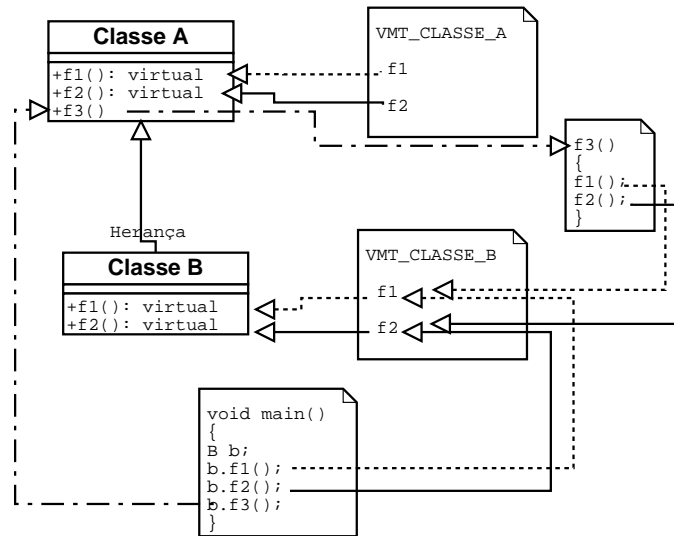
Este mecanismo também recebe o nome de ligação dinâmica, pois o método que vai ser executado, vai ser definido em tempo de execução.

Veja na Figura 16.1 o funcionamento da ligação dinâmica. Toda chamada a `f1` e `f2` passa pela VMT. Quando você cria um objeto do tipo `A`, o mesmo usa a VMT da classe `A` e sempre acessa os métodos da classe `A`. O mesmo é válido para a classe `B`, quando você cria um objeto do tipo `B`, o mesmo usa a VMT da classe `B` e sempre acessa os métodos da classe `B`. Observe que quando `B` acessa o método `f3`, acessa o método da classe `A` porque o mesmo não foi redefinido na classe `B`. Mas quando `f3` acessa `f1` e `f2`, acessa os métodos `f1` e `f2` da classe `B` porque passa pela VMT da classe `B`.

16.2.1 Sentenças para métodos virtuais

- Não devemos chamar métodos virtuais de dentro do construtor. Porque o construtor da classe sempre vai chamar os métodos da própria classe, visto que a VMT da classe derivada ainda não foi criada.
- Uma vez que um método é declarado como `virtual` este o será para sempre e em todas as classes derivadas.

Figura 16.1: Ilustração da ligação dinâmica.



- Os métodos virtuais devem ter exatamente o mesmo protótipo, se o nome for igual e os parâmetros diferentes, teremos sobrecarga de métodos.
- Retorno diferente não diferencia métodos, o que diferencia os métodos é o seu nome e seus parâmetros.
- Métodos virtuais não podem ser estáticos.
- Uma classe abstrata normalmente é uma classe base ou superclasse.
- Uma classe abstrata é usada como uma interface para acessar, através de um ponteiro, as classes derivadas.
- O uso de métodos virtuais em heranças permite a implementação do conceito de polimorfismo e elimina boa parte das estruturas switch.

16.3 Como implementar o polimorfismo

Vimos anteriormente que polimorfismo significa muitas formas, ou seja, para cada classe derivada o método virtual vai ser redefinido, mas de uma forma diferente. Vimos ainda que o método virtual que vai ser efetivamente executado é endereçado pela VMT.

Para implementar o polimorfismo é necessário uma hierarquia de classes com métodos virtuais e que a criação dos objetos seja dinâmica (com o uso de um ponteiro para a classe base). Apresenta-se a seguir um exemplo que esclarece o uso do polimorfismo.

O programa e88-Polimorfismo.cpp usa as classes TPonto, TCirculo e TElipse definidas anteriormente. O programa inicia criando um ponteiro para a classe base da hierarquia, a classe TPonto. Dentro do do..while(), o usuário seleciona qual o tipo de objeto a ser criado, isto é, TPonto, TCirculo ou TElipse. A seguir, cria dentro do switch o objeto selecionado.

Observe a chamada `ptr->Desenha()`; não existe nenhuma referência ou informação sobre qual foi o objeto criado. Mas com o uso do mecanismo de polimorfismo, o método `desenha` que será executado é aquele do objeto que foi criado.

Observe que depois que o objeto for criado, pode-se usar os métodos públicos da classe `TPonto` e estes irão executar o método do objeto que foi criado. Se criar um objeto `TPonto`, executa o método `Desenhar` de `TPonto`. Se criar um objeto `TCirculo`, executa o método `Desenhar` de `TCirculo`, e se criar um objeto `TElipse` executa o método `Desenhar` de `TElipse`.

Veja no final da listagem a seção Saída, observe como fica a saída selecionando-se os diferentes tipos de objetos.

Listing 16.1: Exemplo de uso do polimorfismo.

```
//-----Arquivo e88-Polimorfismo.cpp
/* Cobre polimorfismo */
#include <iostream>
using namespace std;

#include "e87-TPonto.h"
#include "e87-TCirculo.h"
#include "e87-TElipse.h"

//Exemplo de criação e uso do objeto TPonto, TCirculo e TElipse
int main()
{
//1- Crie um ponteiro para a classe base
TPonto * ptr = NULL;

int selecao;
//2- Pergunte para o usuário qual objeto deve ser criado
do
{
cout<<"\nQual objeto criar? ";
cout<<"\nTPonto..... (1)";
cout<<"\nTCirculo..... (2)";
cout<<"\nTElipse..... (3)";
cout<<"\nPara sair?:";
cin >> selecao;
cin.get();

//3- Crie o objeto selecionado.
switch(selecao)
{
case 1: ptr = new TPonto(1,2);          break;
case 2: ptr = new TCirculo(1,2,3);      break;
case 3: ptr = new TElipse(1,2,3,4);    break;
default:ptr = new TCirculo(1,2,3);      break;
}

//4- Agora você pode fazer tudo o que quiser com o objeto criado.
ptr->Desenha ();

//....
//ptr->outros métodos
//....
```

```

//5- Para destruir o objeto criado, use
delete ptr;
ptr = NULL;
} while ( selecao < 4 );

return 0;
}

/*
Para compilar no Linux:
g++ e88-Polimorfismo.cpp e87-TPonto.cpp e87-TCirculo.cpp e87-TElipse.cpp
*/
/*
Novidade:
-----
Uso de polimorfismo
Uso de estrutura de controle switch(){case i: break;}

Uma estrutura switch é usada para seleção de uma opção em diversas,
isto é, switch (opção). A opção a ser executada é aquela
que tem o case valor = opção.

No exemplo abaixo, se o valor de selecao for 1 executa a linha
    case 1: ptr = new TPonto(1,2);          break;

Se o valor de seleção não for nem 1, nem 2, nem 3, executa a opção default.
O break é utilizado para encerrar a execução do bloco switch.

switch(selecao)
{
    case 1: ptr = new TPonto(1,2);          break;
    case 2: ptr = new TCirculo(1,2,3);      break;
    case 3: ptr = new TElipse(1,2,3,4);    break;
    default: ptr = new TCirculo(1,2,3);    break;
}

*/
/*
Saída:
-----
[root@mercurio Cap3-P00UsandoC++]# ./a.out

Qual objeto criar?
TPonto.....(1)
TCirculo.....(2)
TElipse.....(3)
Para sair 4?:0

TPonto: Coordenada x=1
TPonto: Coordenada y=2

TCirculo: Coordenada r1=3

Qual objeto criar?

```



```

TPonto..... (1)
TCirculo..... (2)
TElipse..... (3)
Para sair 4?:1

```

```

TPonto: Coordenada x=1
TPonto: Coordenada y=2

```

```

Qual objeto criar?
TPonto..... (1)
TCirculo..... (2)
TElipse..... (3)
Para sair 4?:2

```

```

TPonto: Coordenada x=1
TPonto: Coordenada y=2

```

```

TCirculo: Coordenada r1=3

```

```

Qual objeto criar?
TPonto..... (1)
TCirculo..... (2)
TElipse..... (3)
Para sair 4?:3

```

```

TPonto: Coordenada x=1
TPonto: Coordenada y=2

```

```

TCirculo: Coordenada r1=3

```

```

TElipse: Coordenada r2=4

```

```

Qual objeto criar?
TPonto..... (1)
TCirculo..... (2)
TElipse..... (3)
Para sair 4?:4

```

```

TPonto: Coordenada x=1
TPonto: Coordenada y=2

```

```

TCirculo: Coordenada r1=3
*/

```

Releia o exemplo com cuidado e observe que depois que o objeto é criado dentro do switch, o mesmo pode ser usado (funções `ptr->desenhar()`, `ptr->outras_funcoes();`), mas você não sabe qual foi o objeto que o usuário criou.

É um mecanismo de programação sensacional, você escreve boa parte do código de forma genérica (`ptr->Desenhar();`), sem se preocupar se foi criado um ponto, um circulo ou uma elipse.

16.3.1 Sentenças para polimorfismo

- Todo objeto criado, sabe a qual classe pertence e usa a VMT da sua classe.

- O mecanismo virtual existe para permitir que as operações sobre um objeto sejam realizadas sem o conhecimento do tipo do objeto que vai ser criado.
- O polimorfismo deixa o programa mais simples e genérico.

16.4 Métodos virtuais puros (Classes abstratas)²10.1

Métodos virtuais puros são métodos que são declarados como virtuais na classe base e que não são implementados. Veja no protótipo que o método é igualado ao valor 0.

Protótipo:

```
class Nome
{
    virtual tipo nomeMétodo(parâmetros)=0;
};
```

16.4.1 Sentenças para métodos virtuais puros (classes abstratas)

- Se você tem uma classe com um ou mais métodos virtuais puros você tem uma classe abstrata.
- Uma classe abstrata não pode criar objetos¹, mas pode ser usada para criar ponteiros.
- Se a classe herdeira não implementa o método puro, a classe herdeira também é uma classe pura, que não pode criar objetos.

16.5 Exemplo completo com polimorfismo

Apresenta-se a seguir um exemplo completo, com um conjunto de classes e o uso de polimorfismo. A hierarquia é ilustrada na Figura 16.2. Observe que não é uma modelagem perfeita, afinal de contas, uma pessoa tem um nome mas não tem uma matrícula. O atributo matrícula foi movido para TPessoa na etapa de Projeto. Da mesma forma, o atributo string Universidade não está bem colocado. Fica para o estudante a tarefa de melhorar esta hierarquia, e implementar no código as modificações que achar pertinentes.

A classe TPessoa.

Listing 16.2: Arquivo TPessoa.h.

```
#ifndef TPessoa_h
#define TPessoa_h

//-----Arquivo
    TPessoa.h
//-----Bibliotecas C/
    C++
#include <fstream>          //vou usar objetos fstream (saida disco,tela,..)
```

¹Como o método só foi declarado e não definido, não posso criar um objeto porque ele poderia chamar um método que não existe.

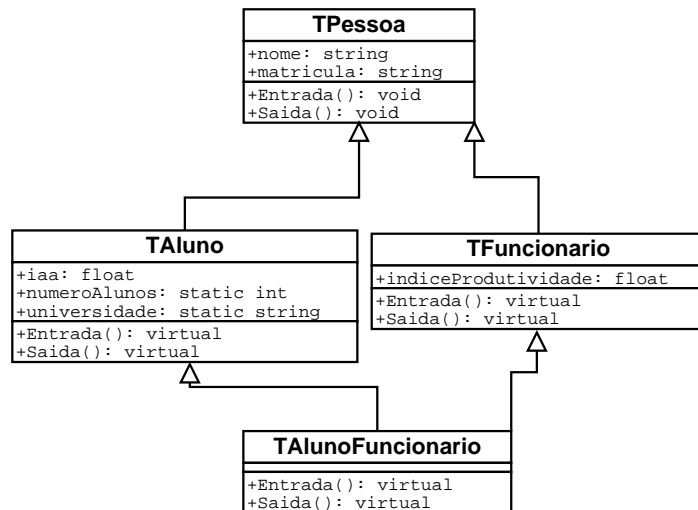


Figura 16.2: Hierarquia TPessoa, TAluno, TFuncionario, TAlunoFuncionario.

```

#include <string>          //vou usar objetos string's

//-----Classe
/*
A classe TPessoa representa uma pessoa (um aluno, um professor, um funcionário)
.
Tem um nome, uma matricula. E métodos básicos para entrada e saída de dados.
Tem alguns construtores e um destrutor
*/
class TPessoa
{
//-----Atributos
//Acesso privado (somente nesta classe)
private:
    std::string nome;
    std::string matricula;

//Acesso público (tendo um objeto pode acessar as funções public)
public:

//-----Métodos
    Construtores
//Declaração dos métodos
//Constrõe objeto (chamada automaticamente na construção do objeto)
    TPessoa();

//Construtor de cópia (Cria uma cópia de um objeto existente)
    TPessoa(const TPessoa& obj);

//Construtor sobrecarregado (com parâmetros)
    TPessoa(std::string _nome, std::string _matricula);

//-----Método
    Destrutor

```

```

//Destrõe objeto (Chamada automaticamente na destruição do objeto)
virtual ~TPessoa();

//-----Métodos
//Método do objeto, altera as propriedades do objeto
//Leitura dos atributos (nome, matricula)
virtual void Entrada();

//Saida dos atributos (nome, matricula)
virtual void Saida(std::ostream &os) const;

//-----Métodos Get /
    Set
//Funções Get
string Getnome() const {return nome;};
string Getmatricula() const {return matricula;};

//Funções Set
void Setnome(std::string _nome) {nome=_nome;}
void Setmatricula(std::string _m) {matricula=_m;}

//Acesso nesta classe e nas classes herdeiras (filhas, subclasses)
protected:
    //.....
};
#endif

```

Listing 16.3: Arquivo TPessoa.cpp.

```

//-----Arquivo TPessoa.
    cpp
//posso usar cout direto
using namespace std;

//inclue arquivo com declaração da classe
#include "TPessoa.h"

//Construtor default (sem parâmetros)
//          /--definição dos valores iniciais de nome e matricula
TPessoa::TPessoa(): nome(""),matricula("")
{
    //Posso inicializar os atributos nome e matricula como acima
    //TPessoa::TPessoa(): nome(0),matricula(0)
    //ou dentro do bloco {...} do construtor, como ilustrado abaixo
    //nome=0;
    //matricula=0;
    cout<<"criou objeto TPessoa construtor default"<<endl;
};

//Construtor de cópia
//Cria uma cópia do objeto, copia todos os atributos (nome,matricula)
TPessoa::TPessoa(const TPessoa& obj)
{
    nome = obj.nome;
    matricula = obj.matricula;
    cout<<"criou objeto TPessoa construtor de cópia"<<endl;
};

```

```

    }

//Construtor sobrecarragado (com parâmetros)
TPessoa::TPessoa(string _nome, string _matricula)
    :nome(_nome),matricula(_matricula)
    {
    //nome = _nome;
    //matricula = _matricula;
    cout<<"criou objeto TPessoa construtor sobrecarregado"<<endl;
    }

//Destrõe objeto
TPessoa::~TPessoa()
    {
    cout<<"destruiu objeto TPessoa"<<endl;
    };

void TPessoa::Entrada()
{
    cout << "Entre com o nome: ";
    getline(cin,nome);

    cout << "Entre com a matricula: ";
    getline(cin,matricula);
}

//Saída de dados
void TPessoa::Saida(ostream &os) const
{
    os << "Nome: " << nome << endl;
    os << "Matricula: " << matricula << endl;
}

```

A classe TAluno.

Listing 16.4: Arquivo TAluno.h.

```

#ifndef TAluno_h
#define TAluno_h

//-----Arquivo TAluno
.h
//-----Bibliotecas C/
C++
#include <fstream>
#include <string>

//inclusão para herança
#include "TPessoa.h"

//-----Classe
/*
A classe TAluno é herdeira da classe TPessoa representa um aluno da
universidade.
E redefine as funções Entrada/Saida.
Adiciona o atributo iaa e os métodos Getiaa(), Setiaa().

```

```

*/
//      /---Nome da classe
//      /      /-Tipo herança
//      /      /      /- nome da classe base
class TAluno: /*virtual */ public TPessoa
{
//-----Atributos
//Acesso privado
private:

    //Atributo normal é criado para cada objeto
    double iaa;

    //Atributo estático é criado na classe
    static int numeroAlunos;

    //Atributo da classe e constante
    static const string universidade;

//Acesso público (tendo um objeto pode acessar as funções public)
public:

//-----Métodos Construtores
    //Construtor default
    TAluno ();

    //Construtor de cópia
    TAluno (const TAluno & obj);

    //Construtor sobrecarregado (com parâmetros)
    TAluno (string _nome, string _matricula, double _iaa = 0);

//-----Método
    //Destrutor
    //Destroe objeto
    virtual ~ TAluno ();

//-----Métodos
    //Leitura dos atributos (nome, matricula)
    virtual void Entrada ();

    //Saida dos atributos (nome, matricula, iaa)
    virtual void Saida (ostream & os) const;

//-----Métodos Get /
    //Set
    //Funções Get
    double Getiaa () const
    {
        return iaa;
    }

    //Funções Set
    void Setiaa (double _iaa)
    {

```

```

    iaa = _iaa;
}

//-----Métodos
    Estáticos
//Funções static podem ser chamados sem um objeto
//e só podem manipular atributos static
    static int GetnumeroAlunos ()
    {
        return numeroAlunos;
    };
    static const string Getuniversidade ()
    {
        return universidade;
    };
};

#endif
/*
PS:
Observe que os métodos que serão herdados tem a palavra chave virtual
na frente da declaração da função.
Ex:
    virtual void Entrada();

A palavra chave virtual informa que esta função pode
ser redefinida na classe herdeira.
*/

```

Listing 16.5: Arquivo TAluno.cpp.

```

//-----Arquivo TAluno
    .cpp
using namespace std;

#include "TAluno.h"

//Atributo estático é aquele que pertence a classe e não ao objeto
//e precisa ser definido depois da classe.
int TAluno::numeroAlunos=0;
const string TAluno::universidade="Universidade_Federal_de_Santa_Catarina";

//Constrõe objeto
//Chamada automaticamente na construção do objeto
TAluno::TAluno() : iaa(0.0)
{
    numeroAlunos++;
    cout<<"criou_objeto_TAluno_("<<numeroAlunos<<")_construtor_default"<<
        endl;
};

//Construtor de cópia
//Cria uma cópia de um objeto existente
TAluno::TAluno(const TAluno& obj) : TPessoa(obj)
{

```

```

        iaa =          obj.iaa;
        numeroAlunos++;
        cout<<"criou objeto TALuno ("<<numeroAlunos<<") construtor de cópia"<<
            endl;
    }

//Construtor sobrecarragado (com parâmetros)
//Observe que iaa tem um inicializador
TAluno::TAluno(string _nome, string _matricula, double _iaa=0)
    : TPessoa(_nome,_matricula),iaa(_iaa)
    {
        numeroAlunos++;
        cout<<"criou objeto TALuno ("<<numeroAlunos
            <<") construtor sobrecarregado"<<endl;
    }

//-----Método
    Destrutor
//Destrõe objeto
TAluno::~TAluno()
    {
        cout<<"destruiu objeto TALuno:"<<numeroAlunos<<endl;
        numeroAlunos--;
    };

//Método redefinido nesta classe
void TAluno::Entrada()
{
    //Chama Método da classe base (para entrada do nome e matricula)
    TPessoa::Entrada();

    //Adiciona aqui o que é diferente nesta classe
    cout << "Entre com o IAA do aluno: ";
    cin>> iaa;
    cin.get();
}

//Método redefinido
void TAluno::Saida(ostream &os) const
{
    TPessoa::Saida(os);
    os << "iaa: " << iaa << endl;
}

```

A classe TFuncionário.

Listing 16.6: Arquivo TFuncionario.h.

```

#ifndef TFuncionario_h
#define TFuncionario_h

//-----Arquivo
    TFuncionario.h
//-----Bibliotecas C/C++
#include <fstream>
#include <string>

```



```

#include "TPessoa.h"      //inclusão para herança

//-----Classe
/*
A classe TFuncionario é herdeira da classe TPessoa
representa um funcionario de uma empresa.
E redefine as funções Entrada/Saida.
Adiciona o indiceProdutividade e métodos GetindiceProdutividade(),
SetindiceProdutividade().
*/
class TFuncionario : /*virtual*/ public TPessoa
{
//-----Atributos
//Acesso privado
private:

    //Atributo normal é criado para cada objeto
    double indiceProdutividade;

//Acesso público (tendo um objeto pode acessar as funções public)
public:

//-----Métodos Construtores
    //Construtor default
    TFuncionario();

    //Construtor de cópia
    TFuncionario(const TFuncionario& obj);

    //Construtor sobrecarragado (com parâmetros)
    TFuncionario(string _nome, string _matricula, double _indiceProdutividade=0);

//-----Método Destrutor
    //Destroe objeto
    virtual ~TFuncionario();

//-----Métodos
    //Leitura dos atributos (nome, matricula)
    virtual void Entrada();

    //Saida dos atributos (nome, matricula, indiceProdutividade)
    virtual void Saida(ostream &os) const;

//-----Métodos Get / Set
    //Funções Get
    double GetindiceProdutividade() const {return indiceProdutividade;}

    //Funções Set
    void SetindiceProdutividade(double _indiceProdutividade)
        {indiceProdutividade=_indiceProdutividade;}

};

#endif

```

Listing 16.7: Arquivo TFuncionario.cpp.

```

//-----Arquivo
    TFuncionario.cpp
using namespace std;

#include "TFuncionario.h"

//Constrõe objeto
//Chamada automaticamente na construção do objeto
TFuncionario::TFuncionario() : indiceProdutividade(0.0)
{
    cout<<"criou_objeto_TFuncionario_constructor_default"<<endl;
};

//Construtor de cópia
//Cria uma cópia de um objeto existente
TFuncionario::TFuncionario(const TFuncionario& obj) : TPessoa(obj)
{
    indiceProdutividade = obj.indiceProdutividade;
    cout<<"criou_objeto_TFuncionario_constructor_de_cópia"<<endl;
}

//Construtor sobrecarregado (com parâmetros)
//Observe que indiceProdutividade tem um inicializador
TFuncionario::TFuncionario(string _nome, string _matricula,
                           double _indiceProdutividade=0)
: TPessoa(_nome,_matricula),indiceProdutividade ( _indiceProdutividade)
{
    cout<<"criou_objeto_TFuncionario_constructor_sobrecarregado"<<endl;
}

//-----Método Destrutor
//Destroe objeto
TFuncionario::~TFuncionario()
{
    cout<<"destruiu_objeto_TFuncionario:"<<endl;
};

//Método redefinido nesta classe
void TFuncionario::Entrada()
{
    //Chama método da classe base (para entrada do nome e matricula)
    TPessoa::Entrada();

    //Adiciona aqui o que é diferente nesta classe
    cout << "Entre_com_o_indiceProdutividade_do_funcionario: ";
    cin>> indiceProdutividade;
    cin.get();
}

//Método redefinido
void TFuncionario::Saida(ostream &os) const
{
    TPessoa::Saida(os);
}

```

```

        os << "indiceProdutividade_␣:␣"    << indiceProdutividade << endl;
    }

```

A classe TAlunoFuncionário.

Listing 16.8: Arquivo TAlunoFuncionario.h.

```

#ifndef TAlunoFuncionario_h
#define TAlunoFuncionario_h

//-----Arquivo
    TAlunoFuncionario.h
//-----Bibliotecas C/C++
#include <fstream>        //vou usar objetos fstream (saida disco,tela,..)
#include <string>         //vou usar objetos string's

#include "TAluno.h"
#include "TFuncionario.h"

//-----Classe
/*
A classe TAlunoFuncionario representa uma pessoa
(um aluno, um professor, um funcionário).
Tem um nome, uma matricula. E métodos básicos para entrada e saída de dados.
Tem alguns construtores e um destrutor
*/
//          /-Nome Da classe (esclarecedor)
//          /                               /-Primeira herança
//          /                               /                               /-Segunda herança
class TAlunoFuncionario: public TAluno, public TFuncionario
{
//-----Atributos
//índice de pobreza
double ip;

public:
//-----Métodos Contrutores
    TAlunoFuncionario();

//-----Método Destrutor
    virtual ~TAlunoFuncionario();

//-----Métodos
//Métodos virtuais das classes base que são redeclarados
//devem ser redefinidos.
    virtual void Entrada();
    virtual void Saida(std::ostream &os) const;

    void Setip(double _ip) {ip=_ip;}
    double Getip() const {return ip;}
};
#endif

/*
Novidade: Herança múltipla
*/

```

Listing 16.9: Arquivo TAlunoFuncionario.cpp.

```

//-----Arquivo TAlunoFuncionario.cpp
using namespace std;

#include "TAlunoFuncionario.h"

TAlunoFuncionario::TAlunoFuncionario()
{
    cout<<"criou objeto TAlunoFuncionario construtor default"<<endl;
};

//Destroe objeto
TAlunoFuncionario::~TAlunoFuncionario()
{
    cout<<"destruiu objeto TAlunoFuncionario"<<endl;
};

void TAlunoFuncionario::Entrada()
{
    TAluno::Entrada();
    TFuncionario::Entrada();
    cout << "Entre com o indice de pobreza: ";
    cin>>ip;
    cin.get();
}

/*
void TAlunoFuncionario::Entrada()
{
    //Solução para não chamar nome e matrícula 2 vezes
    //Entrada de nome, matricula, iaa
    TAluno::Entrada();

    //Entrada do indiceProdutividade
    cout << "Entre com o indiceProdutividade do funcionario: ";
    cin>> indiceProdutividade;
    cin.get();

    //Entrada do indicepobreza (ip)
    cout << "Entre com o indice de pobreza: ";
    cin>>ip;
    cin.get();
}
*/

//Saída de dados
void TAlunoFuncionario::Saida(ostream &os) const
{
    TAluno::Saida(os);
    TFuncionario::Saida(os);
    os << "indice_pobreza=: " << ip << endl;
}

```

Implementação, exemplo e91-class-Heranca.cpp, utiliza as classes acima definidas.

Listing 16.10: Arquivo e91-class-Heranca.cpp.

```

//-----Arquivo main.
    cpp
#include <fstream>
#include <string>
#include <vector>
#include "TPessoa.h"    //inclusão para uso
#include "TAluno.h"    //inclusão para uso

using namespace std;

//Uso de variável e função global deve ser evitado.
void RelacaoPessoalUniversidade(ostream &os,vector< TAluno > aluno);

int main ()
{
    cout<<"\a\nPrograma_e91"<<endl;
    string linha="-----\n
        ";

    //Cria um objeto professor do tipo TPessoa
    TPessoa professor;

    cout << "Entre_com_o_nome_do_professor: ";
    string nome;
    getline(cin,nome);
    professor.Setnome(nome);

    cout << "Entre_com_a_matricula_do_professor: ";
    string matricula;
    getline(cin,matricula);
    professor.Setmatricula(matricula);

    int numeroAlunos;
    cout<<"Entre_com_o_numero_de_alunos_da_disciplina(ex=3):";
    cin >> numeroAlunos;
    cin.get();
    //Cria um array de objetos alunos do tipo TPessoa
    vector< TAluno > aluno(numeroAlunos);

    cout<<linha;
    for(int contador=0; contador < numeroAlunos; contador++)
    {
        cout << "Aluno_"<< contador <<endl;
        aluno[contador].Entrada();
    }

    //Saída de dados
    RelacaoPessoalUniversidade(cout,aluno);

    //Uso construtor de copia
    cout << linha;
    cout << linha;
    cout<< " _executando:_TPessoa_professor2(professor);_"<<endl;

```

```

TPessoa professor2(professor);
professor2.Saida(cout);

//Uso construtor de cópia pela atribuição
{
cout << linha;
cout << linha;
cout<< "┐executando┐:TPessoa┐professor3┐=┐professor2;"<<endl;
TPessoa professor3 = professor2;          ///-Cria objeto professor3
professor3.Saida(cout);

//Acessando funções Get do objeto diretamente
cout<<"\nUsando┐funcoes┐objeto.get┐diretamente"<<endl;
cout<<"\np3.Getnome()="<<professor3.Getnome();
cout<<"\np3.Getmatricula()="<<professor3.Getmatricula()<<endl;
}          ///-Destroe professor3

//Uso construtor sobrecarregado
cout << linha;
cout << linha;
cout<< "┐executando┐:TPessoa┐professor4(nome,matricula);"<<endl;
nome = "Jose┐A.Belini┐Da┐Cunha┐Neto";
matricula="21-5-1980";
TPessoa professor4(nome,matricula); ///iaa usa o default
professor4.Saida(cout);

//Uso construtor de cópia pela atribuição (usando objetos da herança)
{
cout << linha;
cout << linha;
cout<< "┐executando┐:TPessoa┐professor5┐=┐aluno[0];"<<endl;
TPessoa professor5 = aluno[0]; ///-Cria objeto professor5
professor5.Saida(cout);
}          ///-Destroe professor5

//Acesso a disco (vai criar arquivo "lista_disciplina.dat")
//e chamar a função de saída com a relação do pessoal
ofstream fout("lista_disciplina.dat");///,ios::out);
RelacaoPessoalUniversidade(fout,aluno);
fout.close();

//delete aluno[];
cin.get();
return 0;
}

void RelacaoPessoalUniversidade(ostream &os,vector< TAluno > aluno)
{
string linha="-----\n"
;
os << linha<< "Relação┐de┐Pessoal┐da┐" << TAluno::Getuniversidade()
<< "\n"<<linha << endl;
for(int contador=0; contador < aluno.size(); contador++)

```

```

    {
        os << linha;
        os << "Aluno_" << contador<<endl;
        aluno[contador].Saida(os);
    }

    //Novidade, acesso de uma função estática (da classe) sem um objeto.
    os << linha;
    os << TAluno::Getuniversidade()<<endl;
    os<<"Número_de_alunos="<<TAluno::GetnumeroAlunos()<<endl;
}

```

```

/*
Novidades:
-----
-Declaração e definição de uma herança
-Especificação de acesso (public, protected, e private)
-Uso de uma função global dentro de main, que recebe um
objeto ostream, a saída pode ser para tela ou para disco.
*/
/*
Observe que TPessoa::GetnumeroAlunos() funciona quase como
uma função global, pois posso acessar de qualquer lugar.
*/
/*
Saída:
-----
[andre2@mercurio e91-heranca]$ ./a.out
criou objeto TPessoa construtor default
Entre com o nome do professor: Andre Bueno
Entre com a matricula do professor: 11111111
Entre com o número de alunos da disciplina (ex =3):3
criou objeto TPessoa construtor default
criou objeto TAluno (1) construtor default
criou objeto TPessoa construtor default
criou objeto TAluno (2) construtor default
criou objeto TPessoa construtor default
criou objeto TAluno (3) construtor default
-----
Aluno 0
Entre com o nome do aluno: Pedro Bo
Entre com a matricula do aluno: 22222
Entre com o IAA do aluno: 2
Aluno 1
Entre com o nome do aluno: Saci Perere
Entre com a matricula do aluno: 33333
Entre com o IAA do aluno: 3
Aluno 2
Entre com o nome do aluno: Mula Sem Cabeça
Entre com a matricula do aluno: 4444
Entre com o IAA do aluno: 4
-----
Relação de Pessoal da Universidade Federal de Santa Catarina

```

```

-----
-----
Aluno 0
Nome do aluno: Pedro Bo
Matricula : 22222
iaa : 2
-----
Aluno 1
Nome do aluno: Saci Perere
Matricula : 33333
iaa : 3
-----
Aluno 2
Nome do aluno: Mula Sem Cabeça
Matricula : 4444
iaa : 4
-----
Universidade Federal de Santa Catarina
Número de alunos = 3
-----
    executando : TPessoa professor2(professor);
criou objeto TPessoa construtor de cópia
Nome do aluno: Andre Bueno
Matricula : 11111111
-----
    executando : TPessoa professor3 = professor2;
criou objeto TPessoa construtor de cópia
Nome do aluno: Andre Bueno
Matricula : 11111111
-----
Usando funcoes objeto.get diretamente

p3.Getnome()=Andre Bueno
p3.Getmatricula()=11111111
destruiu objeto TPessoa          //professor3
-----
    executando : TPessoa professor4(nome,matricula);
criou objeto TPessoa construtor sobrecarregado
Nome do aluno: Jose A.Belini Da Cunha Neto
Matricula : 21-5-1980
-----
    executando : TPessoa professor5 = aluno[0];
criou objeto TPessoa construtor de cópia
Nome do aluno: Pedro Bo
Matricula : 22222
destruiu objeto TPessoa          //professor5

destruiu objeto TPessoa          //professor4
destruiu objeto TPessoa
destruiu objeto TAluno:3        //Mula Sem Cabeça

```



```

destruiu objeto TPessoa
destruiu objeto TAluno:2           //Saci Perere
destruiu objeto TPessoa
destruiu objeto TAluno:1           //Pedro Bo
destruiu objeto TPessoa           //
destruiu objeto TPessoa           //professor
*/
/*
Observe que quando cria TPessoa executa o construtor de TPessoa,
quando cria TAluno executa primeiro o construtor de TPessoa
e depois de TAluno.
*/

```

Implementação, exemplo e92-class-Heranca-e-Polimorfismo.cpp, utiliza as classes acima definidas.

Listing 16.11: Arquivo e92-class-Heranca-e-Polimorfismo.cpp.

```

//-----Arquivo main.cpp
#include <fstream>
#include <string>

#include "TPessoa.h"           //inclusão para uso
#include "TAluno.h"           //inclusão para uso

using namespace std;

int
main ()
{
    cout << "\a\nPrograma_e92" << endl;

    string linha =
        "-----\n";
    int resp = 0;
    do
    {
        cout << linha <<
            "Seleção_do_tipo_de_objeto_(0=TPessoa)(1=TAluno)(-1 para sair):";
        cin >> resp;
        cin.get ();

        //Cria um ponteiro para um objeto, pode ser um TPessoa ou um TAluno
        //o ponteiro sempre aponta para a classe base.
        TPessoa *pobj = 0;

        //Estrutura de controle
        switch (resp)
        {
            case -1:
                cout << "\nSair." << endl;
                break;
            case 0:           //Cria objeto TPessoa
                pobj = new TPessoa ();
                break;

            case 1:           //Cria objeto TAluno

```

```

    default:
        pobj = new TAluno ();
        break;
    }
    //não criou objeto selecionado por falta de memória, ou
    //porque selecionou sair
    if (pobj == 0)
        break;                               //sai do (do...while)

    //Daqui para baixo usa pobj sem saber se criou TPessoa ou TAluno
    pobj->Entrada ();
    pobj->Saida (cout);
    delete pobj;
}
while (resp != -1);
cin.get ();
return 0;
}

/*
Novidades:
-----
Uso do conceito de polimorfismo.
1-Declara ponteiro para classe base e zera o mesmo
2-Solicita ao usuário a opção desejada
3-Cria objeto selecionado (com new..)
4-Usa o objeto normalmente (pobj->Entrada());....
5-Destrói o objeto

Observe que a partir de 3, não sabe qual objeto foi criado.
Mas acessa os métodos do objeto criado normalmente.
*/
/*
Observe que TPessoa::GetnumeroAlunos() funciona quase como uma função global,
pois posso acessar de qualquer lugar.
*/
/*
Saída:
-----
[andre2@mercurio e95]$ ./a.out
-----
Seleção do tipo de objeto (0=TPessoa)(1=TAluno)(-1 para sair):0
criou objeto TPessoa construtor default
Entre com o nome: Joao Borges Laurindo
Entre com a matrícula: 12-23112-90
Nome : Joao Borges Laurindo
Matrícula : 12-23112-90
destruiu objeto TPessoa
-----
Seleção do tipo de objeto (0=TPessoa)(1=TAluno)(-1 para sair):1
criou objeto TPessoa construtor default
criou objeto TAluno (1) construtor default
Entre com o nome: Celso Peres Fernandes
Entre com a matrícula: 13-45-11
Entre com o IAA do aluno: 4

```

```

Nome : Celso Peres Fernandes
Matricula : 13-45-11
iaa : 4
destruiu objeto TAluno:1
destruiu objeto TPessoa
-----
Seleção do tipo de objeto (0=TPessoa)(1=TAluno)(-1 para sair):-1
Sair.
*/

```

Implementação, exemplo e93-class-Heranca-Multipla.cpp, utiliza as classes acima definidas.

Listing 16.12: Arquivo e93-class-Heranca-Multipla.cpp.

```

//-----Arquivo main.cpp
#include <fstream>
#include <string>

#include "TPessoa.h"           //inclusão de todos os arquivos
#include "TAluno.h"           //de cabeçalho *.h
#include "TFuncionario.h"     //que vou usar.
#include "TAlunoFuncionario.h"

using namespace std;

int main ()
{
    cout<<"\a\nPrograma_e93"<<endl;

    string linha="-----\n"
        ;
    int resp=0;
    do
        {
            cout    << linha
                <<"Seleção do tipo de objeto\n\a"
                <<"TPessoa.....0\n"
                <<"TAluno.....1\n"
                <<"TFuncionario.....2\n"
                <<"TAlunoFuncionario.....3:\n"
                <<linha;

            cin>>resp;
            cin.get();

            //Cria um ponteiro para um objeto, pode ser um TPessoa ou um TAluno
            //o ponteiro sempre aponta para a classe base.
            TPessoa* pobj=0;

            //Estrutura de controle
            switch(resp)
            {
                case 0: //Cria objeto TPessoa
                    pobj = new TPessoa();
                    break;
                case 1: //Cria objeto TAluno
                    pobj = new TAluno();

```

```

        break;
    case 2: //Cria objeto TFuncionario
        pobj = new TFuncionario();
        break;
    case 3: //Cria objeto TAlunoFuncionario
        //pobj = new TAlunoFuncionario();
        {
            TAlunoFuncionario paf;
            paf.Entrada();
            paf.Saida(cout);
        }
        break;

    case -1:
    default:
        cout<<"\nSair."<<endl;
        break;
        break;
}

if(pobj != 0)
{
    //Daqui para baixo usa pobj sem saber se criou TPessoa,
    //TAluno, TFuncionario
    pobj->Entrada();
    pobj->Saida(cout);
    delete pobj;
}

while(resp != -1);
cin.get();
return 0;
}

/*
Novidades:
-----
-Uso do conceito de polimorfismo
-Uso de objetos com herança múltipla
*/

/*
Saída: (para herança múltipla normal)
-----
[andre2@mercurio e91_heranca-e92_polimorfismo-e93_herancamultipla]$ ./e93

Programa e93
-----
Seleção do tipo de objeto
TPessoa.....0
TAluno.....1
TFuncionario.....2
TAlunoFuncionario.....3:
-----

```

```

0
criou objeto TPessoa construtor default
Entre com o nome: nome da pessoa
Entre com a matricula: 1111111111
Nome : nome da pessoa
Matricula : 1111111111
destruiu objeto TPessoa
-----
Seleção do tipo de objeto
TPessoa.....0
TAluno.....1
TFuncionario.....2
TAlunoFuncionario.....3:
-----
1
criou objeto TPessoa construtor default
criou objeto TAluno (1) construtor default
Entre com o nome: Aluno fulano de tal
Entre com a matricula: 2222312
Entre com o IAA do aluno: 3
Nome : Aluno fulano de tal
Matricula : 2222312
iaa : 3
destruiu objeto TAluno:1
destruiu objeto TPessoa
-----
Seleção do tipo de objeto
TPessoa.....0
TAluno.....1
TFuncionario.....2
TAlunoFuncionario.....3:
-----
2
criou objeto TPessoa construtor default
criou objeto TFuncionario construtor default
Entre com o nome: Funcionario padrao
Entre com a matricula: 2-5ds-rst
Entre com o indiceProdutividade do funcionario: 0.78
Nome : Funcionario padrao
Matricula : 2-5ds-rst
indiceProdutividade : 0.78
destruiu objeto TFuncionario:
destruiu objeto TPessoa
-----
Seleção do tipo de objeto
TPessoa.....0
TAluno.....1
TFuncionario.....2
TAlunoFuncionario.....3:
-----
3
criou objeto TPessoa construtor default
criou objeto TAluno (1) construtor default
criou objeto TPessoa construtor default
criou objeto TFuncionario construtor default

```

```

criou objeto TAlunoFuncionario construtor default
Entre com o nome: Jose da Silva Funcionario e Aluno
Entre com a matricula: 44444444444444
Entre com o IAA do aluno: 4
Entre com o nome: Jose da Silva Funcionario e Aluno r
Entre com a matricula: 4545454545
Entre com o indiceProdutividade do funcionario: .75
Entre com o indice de pobreza: .99
Nome : Jose da Silva Funcionario e Aluno
Matricula : 44444444444444
iaa : 4
Nome : Jose da Silva Funcionario e Aluno r
Matricula : 4545454545
indiceProdutividade : 0.75
indice pobreza= : 1073833876
destruiu objeto TAlunoFuncionario
destruiu objeto TFuncionario:
destruiu objeto TPessoa
destruiu objeto TAluno:1
destruiu objeto TPessoa
*/

/*
Saída 2: (para herança múltipla virtual)
-----
Com apenas 2 modificações, a inclusão da palavra chave virtual
nas heranças de TAluno e TFuncionário o programa só cria o TPessoa uma vez.

As modificações:
Herança normal : class TAluno :          public TPessoa
Herança virtual: class TAluno : virtual public TPessoa
Herança normal : class TFuncionario :    public TPessoa
Herança virtual: class TFuncionario : virtual public TPessoa

E a nova saída do programa

[andre2@mercurio e91_heranca-e92_polimorfismo-e93_herancamultipla]$ ./e93

Programa e93
-----
Seleção do tipo de objeto
TPessoa.....0
TAluno.....1
TFuncionario.....2
TAlunoFuncionario.....3:
-----
3
criou objeto TPessoa construtor default
criou objeto TAluno (1) construtor default
criou objeto TFuncionario construtor default
criou objeto TAlunoFuncionario construtor default
Entre com o nome: Adirlei Andre Kraemer
Entre com a matricula: 456654
Entre com o IAA do aluno: 3.879
Entre com o nome: Adirlei Andre Kraemer

```

```

Entre com a matricula: 55555
Entre com o indiceProdutividade do funcionario: 5
Entre com o indice de pobreza: .9
Nome : Adirlei Andre Kraemer
Matricula : 55555
iaa : 3.879
Nome : Adirlei Andre Kraemer
Matricula : 55555
indiceProdutividade : 5
indice pobreza= : 0.9
destruiu objeto TAlunoFuncionario
destruiu objeto TFuncionario:
destruiu objeto TAluno:1
destruiu objeto TPessoa
-----
Seleção do tipo de objeto
TPessoa.....0
TAluno.....1
TFuncionario.....2
TAlunoFuncionario.....3:
-----
-1

Sair.
*/
/*
Dica:
-Observe a ordem de criação e destruição.
-Com a herança múltipla normal os atributos nome
e matricula eram criados 2 vezes
-Com a herança múltipla virtual os atributos nome
e matricula são criados 1 vez
-Observe que mesmo com herança múltipla, esta pedindo
o nome e a matricula 2 vezes.
*/

```

Veremos o uso de arquivos makefile posteriormente (Parte V, Programação para Linux). Mas você pode compilar as listagens acima apresentadas utilizando o arquivo makefile a seguir.

Listing 16.13: Arquivo makefile para exercícios e91, e92, e93.

```

# Um arquivo makefile automatiza a geração de um programa
# Cada arquivo nome.h (declarações) esta ligado a um arquivo nome.cpp (
    definições)
# Cada arquivo nome.cpp depois de compilado gera um arquivo nome.obj
# Diversos arquivos nome.obj são agrupados pelo linker para gerar o programa
    executável
# Diagramaticamente:
#
# (a.h + a.cpp )
#     \----->Compilação   ---> a.obj
# (b.h + b.cpp )
#     \----->Compilação   ---> b.obj
# (main.cpp )
#     \----->Compilação   ---> main.obj
#
#                                     |
#                                     |Linkagem

```

```

#                               \|\|
#                               .
#                               main.exe
#
#Exemplo de arquivo makefile
#Variáveis internas
ARQUIVOS = TAluno.cpp TAlunoFuncionario.cpp TFuncionario.cpp TPessoa.cpp
OBJETOS = TAluno.o TAlunoFuncionario.o TFuncionario.o TPessoa.o

#Arquivos de include do G++ estão em /usr/include/g++
DIRETORIO_INCLUDE = -I/usr/include/g++ -I/usr/include -Iheader -Isource
DIRCL=
DIRETORIO_LIB= -lm
COMPILADOR= g++

#As linhas abaixo especificam as sub-rotinas

#Sub0: all executa todas as subrotinas
all: obj e91 e92 e93

#Subrotina obj: Compila os ARQUIVOS
obj: $(ARQUIVOS)
    $(COMPILADOR) -c $(ARQUIVOS) $(DIRETORIO_INCLUDE)

#subrotina e91: gera executável e91
e91: $(OBJETOS)
    $(COMPILADOR) e91-class-Heranca.cpp $(OBJETOS)
    $(DIRETORIO_INCLUDE) $(DIRETORIO_LIB) -o e91

#subrotina e92: gera executável e92
e92: $(OBJETOS)
    $(COMPILADOR) e92-class-Heranca-e-Polimorfismo.cpp $(OBJETOS)
    $(DIRETORIO_INCLUDE) $(DIRETORIO_LIB) -o e92

#subrotina e93: gera executável e93
e93: $(OBJETOS)
    $(COMPILADOR) e93-class-Heranca-Multipla.cpp $(OBJETOS)
    $(DIRETORIO_INCLUDE) $(DIRETORIO_LIB) -o e93

clean:
    rm *.o
    rm e91 e92 e93

```

Veja a seguir uma listagem com a saída gerada pela execução do programa make, observe que foram incluídos comentários.

Listing 16.14: Saída gerada pelo makefile dos exercícios e91, e92, e93.

Saída do comando make
 =====

```

-----
1)
Quando executo

```



```
make e91
```

O make verifica as dependências para gerar o programa e91, compila os arquivos que são necessários e então usa o linker para gerar o programa e91. Observe na saída abaixo que o make vai compilar TAluno, TFuncionario, TAlunoFuncionario, TPessoa e depois e91_class-Heranca.cpp gerando o programa e91.

```
[andre2@mercurio gnu]$ make e91
g++ -c -o TAluno.o TAluno.cpp
g++ -c -o TAlunoFuncionario.o TAlunoFuncionario.cpp
g++ -c -o TFuncionario.o TFuncionario.cpp
g++ -c -o TPessoa.o TPessoa.cpp
g++ e91_class-Heranca.cpp TAluno.o TAlunoFuncionario.o
    TFuncionario.o TPessoa.o -I/usr/include/g++ -I/usr/include
    -Iheader -I/source -lm -o e91
```

2)

No exemplo abaixo mandei compilar o e92, como já havia gerado os arquivos *.o executou direto o g++ e92_class-Heranca-e-Polimorfismo.cpp.... (só recompilou aquilo que era necessário)

```
[andre2@mercurio gnu]$ make e92
g++ e92_class-Heranca-e-Polimorfismo.cpp TAluno.o TAlunoFuncionario.o
    TFuncionario.o TPessoa.o -I/usr/include/g++
    -I/usr/include -Iheader -I/source -lm -o e92
```


Capítulo 17

Friend

Apresenta-se neste capítulo o uso de classes e métodos friend em C++. Como usar o conceito de friend para obter um maior encapsulamento das diversas classes.

17.1 Introdução ao conteito de friend

A palavra chave friend é utilizada para dar a uma classe ou método a possibilidade de acesso a membros não públicos de uma outra classe.

Lembre-se, se você tem um objeto `obj_a` do tipo A, você só pode acessar os atributos e métodos definidos como públicos na classe A.

Exemplo:

- Um desconhecido seu não tem acesso a sua casa, aos seus bens pessoais. Um amigo seu pode entrar na sua casa e fazer uso de algumas coisas suas. Na programação orientada a objeto funciona da mesma forma. Basta você informar quem são os amigos.

17.2 Classes friend

A declaração friend fornece a um método ou a uma classe, o direito de ser amiga de um outro objeto e de ter acesso aos atributos e métodos do objeto amigo.

Você pode declarar toda a classe ou apenas um determinado método como amigo.

Se uma classe A for amiga da classe B, os métodos da classe A podem acessar os atributos e métodos de um objeto da classe B.

No exemplo a seguir, o método `fA` da classe A, pode acessar os atributos da classe B, porque toda classe A é declarada como amiga da classe B.

Listing 17.1: Usando métodos e classes friend.

```
#include <iostream>

//-----A.h
//Somente declara a classe B
class B;

//Declaração da classe A
```

```

class A
{
private:
//objeto tipo int com nome a
int a;

//Método da classe A que recebe um objeto do tipo B como parâmetro
void fA(B& obj);
};

//-----B.h
#include "A.h"
class C;

//Declaração da classe B
class B
{
private:
int b;

//A classe A é amiga, assim, o método fA pode acessar os atributos de B
friend class A;

//Método da classe B que recebe um objeto do tipo C.
void fB(C& obj);
};

//-----C.h
class C
{
private:
int c;

friend void B::fB(C& obj);
};

//-----A.cpp
#include "A.h"
#include "B.h"
void A::fA(B& obj)
{
a = obj.b;
};

//-----B.cpp
#include "B.h"
#include "C.h"
void B::fB(C& obj)
{
b = obj.c;
};

//-----C.cpp
#include "C.h"

```

```
//-----Main.cpp
void main()
{
    std::cout << "Criando objetos A a; B b; C c;" << std::endl;
    A a;
    B b;
    C c;
}
```

Observe neste exemplo que toda classe B é privada, desta forma os atributos e métodos de B só podem ser acessados pela classe A.

17.3 Métodos friend

No exemplo abaixo, um método da classe B é declarado como amigo da classe C. Este método poderá acessar os atributos e métodos da classe C.

```
Exemplo:
//-----c.h
#include "b.h"
class C
{
    int c;
    /*O método abaixo é amigo desta classe, logo, pode acessar os atributos de C*/
    friend void B::fB(C& );
};
```

Em resumo:

- O método fA pode acessar os atributos de B porque a classe A é amiga de B.
- O método fB, pode acessar os atributos de C porque foi declarado como amigo de C.

```
Exemplo:
//Uma função comum
retorno nomeFunção(parametros);
class A
{
    //Um método comum declarado como friend numa classe
    friend retorno nomeFunção(parametros);
    void FA();
};
class B
{
    //Uma função da classe A declarada como amiga da classe B
    friend A::void FA();
};
```

```
//Uma função template (ou gabarito)1
typename<tipo>
retorno nomeFunção(tipo p);
//Uma função template declarada como friend
class C
{
friend retorno nomeFunção(typename <tipo>);
};
```

17.4 Sentenças para friend

- Se a classe A é amiga de B que é amiga de C, não implica que A é amiga de C.
- A única maneira de especificar o relacionamento friend mútuo entre duas classes é declarar toda a segunda classe como friend da primeira.
- A declaração friend só é válida para a classe em que foi declarada, ou seja, não vale para as classes derivadas.
- A declaração de um método envolve três aspectos:
 1. O método tem acesso aos membros internos.
 2. O método esta no escopo da classe.
 3. O método precisa de um objeto da classe (a exceção dos métodos estáticos).
Um método friend tem apenas o primeiro aspecto.
- Um método declarado como friend em uma classe, deixa claro que o mesmo faz parte de uma estrutura lógica da classe.
- Construtores, destrutores e métodos virtuais não podem ser friend.

¹Funções e métodos template (ou gabarito) serão descritas no capítulo 22.

Capítulo 18

Sobrecarga de Operador

Neste capítulo vamos descrever a sobrecarga de operadores. Quais os operadores que podem ser sobrecarregados, a sobrecarga como função friend e método membro. No final do capítulo apresenta-se alguns exemplos.

18.1 Introdução a sobrecarga de operadores

Quando você definiu uma classe, você definiu um tipo do programador. Você criou um conjunto de atributos e de métodos que fazem sentido para a classe criada.

Se você criou uma classe polinomio pode criar uma função para somar dois polinômios.

Exemplo:

```
//Cria objetos do tipo Polinomio
Polinomio pol_a, pol_b, pol_c;
//A linha abaixo soma A e B e armazena em C
somar (pol_A, pol_B, &pol_C);
```

Embora seja funcional, a notação acima é uma notação típica de um programa em C.

Seria interessante, se você pudesse realizar operações como soma(+), subtração(-) e multiplicação(*) utilizando os operadores usuais. O operador + realizaria a soma de dois polinômios, o operador * realizaria a multiplicação dos polinômios e assim por diante. Veja o exemplo.

```
pol_c = pol_A + pol_B;
```

Infelizmente, as linguagens de programação usuais, como C, não podem modificar a forma como os operadores operam. Mas C++ não é uma linguagem comum, é muito mais, e com C++ você pode sobrecarregar os operadores (+,=,...) de forma que a linha acima possa ser utilizada. Ou seja, em C++ podemos usar uma notação muito mais clara e próxima da matemática¹.

Entretanto, para podermos usar o operador (+) no lugar da chamada da função somar, é preciso sobrecarregar o operador +.

Ao processo de definir como os operadores (+, -, *, /, ...) vão se comportar, chamamos de *sobrecarga de operador*, em que estamos sobrecarregando um determinado operador, para realizar uma operação da forma como esperamos.

¹Observe que C++ e programação orientada a objeto, aproximam fortemente os conceitos físicos e matemáticos dos conceitos computacionais. O que facilita o entendimento e o desenvolvimento de programas.

Sobrecarga de operador é a definição das tarefas que determinado operador realiza sobre uma classe definida pelo programador.

A seguir vamos apresentar os operadores que podem ser sobrecarregados e a implementação da sobrecarga com funções friend e métodos membro.

18.2 Operadores que podem ser sobrecarregados

Antes de apresentarmos a forma utilizada para sobrecarregar os operadores é preciso classificar os operadores.

Os operadores podem ser unários ou binários. São unários quando atuam sobre um único objeto, e binários quando atuam sobre dois objetos.

Veja na Tabela 18.1 os operadores que podem ser sobrecarregados.

Tabela 18.1: Operadores que podem ser sobrecarregados.

Operadores binários:	+ - * / = < > += -= *= /=
	<< >> >>= <<= != <= >=
	++ - % & ^ ! ~ & = ^=
	= & & % = [] () new delete
Operadores unários:	++ , --, !, ~
Operadores unários ou binários:	& , * , +
Não podemos sobrecarregar:	:: . * ?

Não podemos alterar a regra de precedência e associatividade da linguagem (veja Tabela C.1). Isso significa, que na seqüência: $z = x + y$; primeiro vai realizar a soma e depois a igualdade e esta seqüência não pode ser alterada.

Para a realização da sobrecarga de operadores podemos utilizar dois métodos; sobrecarga como função membro, em que uma função da classe é que realiza a sobrecarga ou sobrecarga como função friend, em que uma função global e friend realiza a sobrecarga.

Vamos abordar primeiro a sobrecarga como função friend.

18.3 Sobrecarga de operador como função friend

Uma função friend é uma função que é declarada como sendo amiga de determinada classe. Como ela é amiga da classe, pode acessar os atributos desta classe como se fizesse parte dela. Com os operadores funciona da mesma forma. Veja o protótipo.

Protótipo:

```
//-----Tipo1.h
class Tipo1
{..
//Dentro da classe declara o operador como função friend
//Declaração de sobrecarga de operador unário como função friend
friend Tipo1 operator X (Tipo1 obj1);
```



```

//Declaração de sobrecarga de operador binário como função friend
friend Tipo1 operator X (Tipo1 obj1, Tipo1 obj2);
};
//-----Tipo1.cpp
//Definição de sobrecarga do operador unário (função global)
Tipo1 operator X (Tipo1 obj1)
{.
return (tipo1);
}
//Definição de sobrecarga do operador binário (função global)
Tipo1 operator X (Tipo1 obj1, Tipo1 obj2)
{
return (Tipo1);
}

```

18.4 Sobrecarga de operador como método membro da classe

Outra forma de sobrecarga de operador, é implementar o método de sobrecarga como membro da classe. Sendo um método da classe, pode acessar diretamente os seus atributos (não é necessário passar o objeto como parâmetro).

Observe que sendo método membro da classe, um operador unário não receberá nenhum parâmetro e um operador binário receberá apenas um parâmetro. Veja a seguir o protótipo.

Protótipo:

```

//-----TNome.h
class TNome
{
tipo atributo;
//Declaração de sobrecarga do operador unário
TNome operatorX ();
//Declaração de sobrecarga do operador binário
TNome operatorX (TNome& objeto2);
};
//-----TNome.cpp
//Definição do operador unário como função membro
TNome TNome::operatorX ()
{
return TNome;
}
//Definição do operador binário como função membro
TNome TNome::operatorX (TNome& objeto2)
{
return TNome;
}

```

18.5 Sentenças para sobrecarga de operador

- Tente usar sempre sobrecarga como método membro. A sobrecarga como função friend cria uma função global, que é desaconselhável em um programa orientado a objeto.
- Quando sobrecarregar <<, não envie a saída diretamente para cout, envie para a stream, pois pode-se ter algo como:

```
cout<<"teste "<<objeto<<" ";
```

- Observe que não ocorre sobrecarga de nomes em escopos diferentes.
- Você já deve ter percebido que muitos operadores do C++ são sobrecarregados para os tipos internos. O operador * é utilizado para multiplicação, declaração de ponteiros ou para obter o conteúdo de ponteiros.
- Esteja atento para efeitos como:

```
obj1 = obj2 = obj3;           //encadeamentos
string1 = string2 = "texto"; //tipos diferentes
complex_c = complex_a + double_x; //sobrecarga múltipla
```

- Segundo Stroustrup, "devido a um acidente histórico, os operadores = (atribuição), & (endereço), e sequenciamento (), tem significados pré-definidos quando aplicados a objetos de classes". Você pode impedir o acesso a estes (e qualquer outro operador), declarando os mesmos como private.
- Se uma classe não tem atributos dinâmicos, você não precisa definir os operadores de cópia e atribuição. Caso contrário crie os operadores de cópia e de atribuição.
- Somente sobrecarregue new e delete como métodos estáticos.
- Criar uma sobrecarga para += não significa que você sobrecarregou + e =.
- Operadores não podem ser definidos para receber ponteiros, exceto (=,& ,,).
- Para reduzir o número de sobrecargas, use métodos de conversão.
- O uso do operador de atribuição (a=b) e do construtor de cópia por inicialização (int a=b;) são duas operações diferentes.

```
Exemplo:
class A
{
//construtor
A(tipo B);
//construtor de cópia por inicialização
A(const A& );
//operador de atribuição
A& operador=(const A& );
};
```

- Os operadores de atribuição =, apontador para membro ->, e parênteses (), só podem ser sobrecarregados como métodos membro da classe (não podem ser friend).
- ²O primeiro argumento de um operador sobrecarregado deve ser um lvalue.
- ² Como a função friend não faz parte da classe, pois só é amiga, ela não recebe um ponteiro implícito this. Ou seja, precisamos passar explicitamente o objeto a ser manipulado pela função.
- ³ Uma sobrecarga é uma chamada de função, assim, os parâmetros podem definir o namespace a ser utilizado.

18.6 Usar funções friend ou funções membro ?

Membro

Use função membro para construtores, destrutores, funções de conversão, operadores unários (=, *=, ++). Sempre que alterar o estado do objeto, prefira funções membro.

Como método membro pode usar:

```
obj->funcao(); //sintaxe de C++
```

A sintaxe deixa claro que o programador pode alterar o objeto usando o método.

Friend

Use função friend para sobrecarga de operadores, quando um terceiro objeto é gerado.

Na dúvida fique com método membro, pois a chamada de uma função friend deixa o código com estilo de C.

```
funcao(obj); //sintaxe de C.
```

Apresenta-se a seguir o protótipo e/ou exemplos de sobrecarga de diversos operadores.

18.7 Protótipos de sobrecarga

Protótipo:

```
//Pré-fixado:
```

```
void operator++(int); //sobrecarga para ++obj;
```

```
void operator--(int); //sobrecarga para --obj;
```

Protótipo:

```
//Pós-fixado:
```

```
void operator++(void); //sobrecarga para obj++;
```

```
void operator--(void); //sobrecarga para obj--;
```

Protótipo:

```
X* operator & (); // endereço de
```

```
X operator& (x); //& binário
```

Os exemplos que serão apresentados mostram na prática como usar sobrecarga de operadores.

Reapresenta-se a seguir a classe TPonto, agora com alguns operadores sobrecarregados.

Listing 18.1: Arquivo e89-TPonto.h

```
//----- Arquivo e87-TPonto.h
#ifndef _TPonto_
#define _TPonto_

#include <iostream>

/*
Define a classe TPonto
Define o tipo de usuário TPonto.
*/
class TPonto
{
//Atributos
//controle de acesso
public:

//atributos de objeto
int x;
int y;

//atributo de classe
static int contador;

//Métodos
public:

//Construtor default
TPonto():x(0),y(0)
    {contador++;};

//Construtor sobrecarregado
TPonto(int _x,int _y):x(_x),y(_y){contador++;};

//Construtor de cópia
TPonto(const TPonto& p)
    {
    x = p.x;
    y = p.y;
    contador++ ;
    };

//Destructor virtual
virtual ~TPonto()
    {contador--;};

//Seta ponto
inline void Set(TPonto& p);

//Seta ponto
inline void Set(int & x, int & y);

//Método inline, obtém x
int Getx() const { return x; };
};
```

```

//Método inline, obtém y
inline int Gety() const;

//Método virtual, desenha o ponto
virtual void Desenha();

//Método Estático
static int GetContador();

//Sobrecarga de operadores como método membro
TPonto& operator++(int);
TPonto& operator--(int);
TPonto& operator+ (TPonto& p2);
TPonto& operator- (TPonto& p2);
TPonto& operator= (TPonto& p2);
TPonto& operator+= (TPonto& p2);
TPonto& operator-= (TPonto& p2);

//Sobrecarga como função friend
friend bool operator==(TPonto& p1,TPonto& p2);
friend bool operator!=(TPonto& p1,TPonto& p2);

friend std::ostream& operator<<(std::ostream& out, TPonto& p);
friend std::istream& operator>>(std::istream& in, TPonto& p);
};

#endif
/*
Novidades:
-----
Definição de operadores sobrecarregados, como o operador soma

TPonto& operator+ (TPonto& p2);

Sobrecarga como método membro e como função friend.
*/

```

Listing 18.2: Arquivo e89-TPonto.cpp.

```

//-----Arquivo TPonto.cpp
#include <iostream>
#include "e89-TPonto.h"

//-----Definição atributo estático
//Definição de atributo estático da classe
int TPonto::contador = 0;

//-----Definição Métodos da classe
//Definição dos métodos de TPonto
//Seta valores de x,y , usando ponto p
void TPonto::Set(TPonto& p)
{
    x = p.Getx(); y = p.Gety();
}

//seta valores de x,y do objeto, usando valores x,y passados como parâmetro

```

```

//Como recebe x,y, e dentro da classe também tem um x,y
//usa o ponteiro this para diferenciar, x do objeto do x parâmetro
void TPonto::Set(int & x, int & y)
{
    this->x = x; //uso de this
    this->y = y;
}

//Método get não muda o objeto, sendo então, declarado como const
int TPonto::Getx() const
{
    return x;
}

//Método estático da classe, pode ser chamado sem um objeto
int TPonto::GetContador()
{
    return contador;
}

//Método virtual
void TPonto::Desenha()
{
    std::cout << "\nTPonto:  Coordenada_x=" << x;
    std::cout << "\nTPonto:  Coordenada_y=" << y <<std::endl;
}

//-----Definição Métodos sobrecarregados
//Definição da sobrecarga do operador ++
//Simplesmente incremento x e y
TPonto& TPonto::operator++(int)
{
    this->x++;
    this->y++;
    return *this;
}

//Definição da sobrecarga do operador --
//Simplesmente decremento x e y
TPonto& TPonto::operator--(int)
{
    this->x--;
    this->y--;
    return *this;
}

//Definição da sobrecarga do operador +
TPonto& TPonto::operator+(TPonto& p2)
{
    TPonto *p3 = new TPonto;
    p3->x = this->x + p2.x;
    p3->y = this->y + p2.y;
    return *p3;
}

```

```

//Definição da sobrecarga do operador +=
TPonto& TPonto::operator+=(TPonto& p2)
{
this->x += p2.x;
this->y += p2.y;
return *this;
}

//Definição da sobrecarga do operador -
TPonto& TPonto::operator-(TPonto& p2)
{
TPonto* p3 = new TPonto;
p3->x = this->x - p2.x;
p3->y = this->y - p2.y;
return *p3;
}

//Definição da sobrecarga do operador -=
TPonto& TPonto::operator-=(TPonto& p2)
{
this->x -= p2.x;
this->y -= p2.y;
return *this;
}

//Definição da sobrecarga do operador =
TPonto& TPonto::operator=(TPonto& p2)
{
this->x = p2.x;
this->y = p2.y;
return *this;
}

//-----Definição sobrecarregarga com
função friend
//Definição da sobrecarga do operador ==
bool operator==(TPonto& p1,TPonto& p2)
{
bool resp = (p1.x == p2.x) && (p1.y == p2.y);
return resp;
}

bool operator!=(TPonto& p1,TPonto& p2)
{
return ! (p1 == p2 );
}

std::ostream& operator<<(std::ostream& out,TPonto& p)
{
out << "<<p.x<<","<<p.y<<";
return out;
}

std::istream& operator>>(std::istream& in,TPonto& p)
{

```

```

in >> p.x;
in >> p.y;
in.get();
return in;
}

```

Programa de teste das sobrecargas.

Listing 18.3: Arquivo e89-Programa.cpp.

```

//-----Arquivo e87-Programa.cpp
#include <iostream>
using namespace std;

#include "e89-TPonto.h"

//Exemplo de criação e uso do objeto TPonto, com sobrecarga
int main()
{
    //Teste de TPonto
    int x ;
    int y ;

    //Cria objeto do tipo TPonto com nome p1,p2,p3
    TPonto p1,p2,p3;

    //Usando operador >>
    cout << "Entre com os valores de p1 [x enter y enter]:";
    cin >> p1;

    //Usando operador <<
    cout<<"---p1-->"<< p1<<" ---p2-->"<< p2<<" ---p3-->"<< p3<<endl;

    //Usando operador =
    p2 = p1;
    cout <<"Após p2=p1"<<endl;
    cout<<"---p1-->"<< p1<<" ---p2-->"<< p2<<" ---p3-->"<< p3<<endl;

    //Usando operador ==
    cout <<"Testando p1==p2"<<endl;
    if( p1 == p2 )
        cout <<"p1==p2"<<endl;
    else
        cout <<"p1!=p2"<<endl;

    //Usando operador ++
    p2++;
    cout<<"p2++"<<endl;
    cout<<"---p1-->"<< p1<<" ---p2-->"<< p2<<" ---p3-->"<< p3<<endl;

    //Usando operador =
    cout <<"Fazendo p3=p2++"<<endl;
    p3 = p2++;
    cout<<"---p1-->"<< p1<<" ---p2-->"<< p2<<" ---p3-->"<< p3<<endl;

    //Usando operador !=

```



```

cout <<"Testando p2==p3"<<endl;
if( p2 != p3 )
    cout<<"p2!=p3"<<endl;
else
    cout<<"p2==p3"<<endl;

//Usando operador + e =
p3 = p1 + p2 ;
cout <<"Após p3=p1+p2"<<endl;
cout<<"---p1-->"<< p1<<"---p2-->"<< p2<<"---p3-->"<< p3<<endl;
}

```

```

/*
Para compilar no Linux:
g++ e89-Programa.cpp e89-TPonto.cpp
*/
/*
Saída:
-----
[andre@mercurio Cap3-P00UsandoC++]$ ./a.out
Entre com os valores de p1 [x enter y enter]: 1 2
---p1-->(1,2) ---p2-->(0,0) ---p3-->(0,0)
Após p2 = p1
---p1-->(1,2) ---p2-->(1,2) ---p3-->(0,0)
Testando p1 == p2
p1==p2
p2++
---p1-->(1,2) ---p2-->(2,3) ---p3-->(0,0)
Fazendo p3 = p2++
---p1-->(1,2) ---p2-->(3,4) ---p3-->(3,4)
Testando p2 == p3
p2==p3
Após p3 = p1 + p2
---p1-->(1,2) ---p2-->(3,4) ---p3-->(4,6)
*/

```


Capítulo 19

Implementando Associações em C++

Como visto na Parte I, filosofia de programação orientada a objeto, existem relacionamentos entre objetos distintos, a estes relacionamentos damos o nome de associações ou ligações. Discute-se neste capítulo a implementação de associações usando C++.

19.1 Introdução as associações em C++

De um modo geral, uma associação pode ser implementada de 3 formas.

- Implementar a associação através de ponteiros para os objetos.
- A criação de uma classe que representa a associação, neste caso, serão necessários ponteiros para a classe associação partindo dos objetos ligados.
- Uso de funções friend.

Uma associação pode ser unidimensional, bidimensional e pode ou não ter atributos de ligação.

19.2 Associação sem atributo de ligação

Se a associação for unidimensional e sem atributo de ligação, a mesma pode ser implementada com um ponteiro na classe que faz o acesso. No exemplo a seguir a classe A acessa a classe B.

```
Exemplo:  
class B;  
class A  
{  
    B* ptr_para_b;  
};  
class B  
{  
};
```

Se a associação for bidirecional, serão necessários ponteiros em ambas as classes. No exemplo a seguir a classe A acessa a classe B e a classe B acessa a classe A.

Exemplo:

```
class B;
class A
{
  B* ptr_para_b;
};
class B
{
  A* ptr_para_a;
};
```

Se a ligação entre as duas classes for do tipo um para muitos (cardinalidade um para muitos), então na classe que acessa muitos deve ser implementado um vetor de ponteiros.

Exemplo:

```
/*No caso da relação entre as classes A e B ter uma
cardinalidade um A e N B's.*/
class B;
Class A
{
  vector< B* > vetor_b(N);
};
Class B
{
  A* ptr-para_a;
};
```

Se a ligação tiver uma cardinalidade muitos para muitos, a solução é criar um dicionário. Um dicionário é um objeto com duas listas de ponteiros e que faz a associação correta entre as classes A e B.

19.3 Associação com atributo de ligação

Se a associação tiver atributos de ligação e estes só existirem em razão da associação, então deverá ser criada uma classe de associação. Veja o exemplo.

Exemplo:

```
class ligação;
class A
{
  int a;
  friend Ligação;
public:
  Ligação* ptr_ligação;
};
class B
```

```
{
int b;
friend Ligação;
public:
Ligação* ptr_ligação;
};
class Ligação
{
public:
atributoLigação;
A* ptrA;
B* ptrB;
};
void main()
{
B b;
b->ptr_ligação->atributoLigação;
//b->ptr_ligação->ptrA->a;
}
```

A classe de ligação pode ser declarada inteira como amiga de A e B.

Se uma das classes tiver uma cardinalidade muitos, pode ser implementado um vetor de ponteiros, e se as duas classes tiverem uma cardinalidade muitos, novamente, deve-se utilizar o conceito de dicionário.

Capítulo 20

Conversões

Neste capítulo vamos apresentar a necessidade e o protótipo das conversões entre objetos. A seguir apresenta-se o construtor de conversão, os métodos de conversão, o construtor de conversão em heranças e o uso de `explicit` em construtores. Discute-se ainda o uso do `dynamic_cast`, do `const_cast` e do `reinterpret_cast`.

20.1 Protótipos para conversões

Apresenta-se a seguir o protótipo para definição e uso dos construtores de conversão e dos operadores de conversão.

Protótipo:

```
class Base
{
// uso de construtor com explicit, seção 20.5
explicit Base (parâmetros);
};
class Derivada : public Base
{
//Construtor de conversão, seção 20.3
//Cria objeto da classe derivada a partir de objeto da classe base
Derivada(const Base& obj){};
//Declaração de método de conversão para classwe Base, seção 20.4
operator Base();
//Declaração de método de conversão, converte de um tipo A para outro tipo, seção 20.4
operator Tipo();
};
//Definição do método de conversão
Tipo Derivada:: operator Tipo()
{ //Descrição de como se processa a conversão
return(tipo);
};
...
//Uso de dynamic_cast, seção 20.7
```

```

Base* ptrBsase = new Derivada;
Derivada* ptrDerivada = dynamic_cast <Derivada*> (Base);
//Uso de static_cast , seção 20.8
tipo der;
tipo base = static_cast<base> (der);
//referências e dynamic_cast, seção 20.12
tipo& ref = dynamic_cast<Tipo& >(r);

```

20.2 Necessidade de conversão

Você já sabe que existe uma hierarquia de classes para os tipos numéricos e que existe um sistema de conversão entre os diferentes tipos numéricos. Assim, um inteiro pode ser convertido em double (sem perda) e um double pode ser convertido em inteiro (com perda).

Os tipos definidos pelo programador também podem ser convertidos.

Observe o seguinte exemplo: Digamos que existe uma classe B e uma classe derivada D. Que foi criado o objeto b1 do tipo B e o objeto d1 do tipo D. Que criamos os objetos b2, fazendo B b2 = d1; e um objeto b3, fazendo B b3 = b1;. Como a classe D é descendente da classe B, um objeto do tipo D pode ser convertido para B.

Entretanto, não é possível criar um objeto do tipo D a partir de um objeto do tipo B, (fazendo D d2 = b1;), porque B é menor que D. O exemplo esclarece.

```

Exemplo:
//Define tipo B
class B
{public:
  int x;
};
//Define tipo D
class D:public B
{public:
  int y;
};
main()
{
  //Cria objeto do tipo B com nome b
  B b1;
  D d1;
  B b3 = b1;    //ok
  B b2 = d1;    //ok
  D d2 = b1;    //erro, b não preenche d.
}

```

Na linha

```
B b2 = d1;
```


o código funciona porque d1 tem os atributos x e y e o compilador faz `b2.x = d1.x;` .

Na linha

```
D d2 = b1;
```

b1 só tem o atributo x, e d2 precisa de um x e um y. Como b1 não tem y, o compilador acusa o erro.

A solução para este tipo de problema é o uso de construtores de conversão, descritos a seguir.

20.3 Construtor de conversão²

Um construtor de conversão é um construtor utilizado para construir um objeto do tipo D a partir de um objeto do tipo B. Isto é, recebe um objeto do tipo base e constrói um objeto do tipo derivado. Veja o exemplo.

Exemplo:

```
class D: public B
{
  int y;
  //Construtor de conversão, recebe um B e constrói um D
  D(const& B obj)
  {
    this->x = obj.x;
    this->y = 0;
  }
};
```

Com o construtor de conversão do exemplo acima, resolvemos o problema da linha

```
D d2 = b1;
```

Entretanto, se tivermos

```
B b;
K k;
b = k;
```

O objeto b e o objeto k já foram criados. Neste caso é necessário um método que converta o objeto k em b. Para solucionar este problema usa-se um método de conversão.

20.4 Métodos de conversão (cast)

Um método de conversão é utilizado para converter um objeto de um tipo em outro. Observe que não precisa ser da mesma herança. O protótipo é dado por;

Protótipo:

```
//Declaração de método de conversão
operator Tipo();
```

Observe o uso da palavra chave **operator**, seguida do nome da classe a ser convertida. Para usar o método de conversão, você pode ter uma conversão implícita ou explícita.

Exemplo:

```
B b;
C c;
b = c;           //Implícita, converte de c para b
b = (B) c;      //Explícita com o uso de cast de C
b = static_cast< B > c; //Explícita com uso do cast estático de C++
```

20.5 Conversão explícita nos construtores com explicit²

Em algumas hierarquias, as conversões ocorrem automaticamente. Se você não quer isso ocorra, ou seja, quer ter controle das conversões, use a palavra chave explicit na declaração do construtor. Veja o exemplo;

Listing 20.1: Uso do explicit.

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

class TMatriz
{
public:
int x;

//Construtor
TMatriz (int _x = 10) :x(_x) { } ;

//Construtor com explicit
//explicit TMatriz (int _x = 10) :x(_x) { }
};

void Print(const TMatriz& mat)
{
cout << mat.x <<endl;
}

main()
{
//Cria matriz de 5 elementos
int i = 5;
TMatriz matriz(i);

//Imprime a matriz
cout << "Print(matriz);␣" ;
Print(matriz);

/*
Abaixo deveria ocorrer um erro, pois Print recebe um int.
Mas como existe um construtor para TMatriz que
recebe um int, vai criar um objeto matriz
```

```
e chamar Print.
*/
cout<<"Print(10);␣";
Print(10);
}

/*
Saída sem uso de explicit:
-----
[andre@mercurio Cap3-P00UsandoC++]$ ./a.out
Print(matriz); 5
Print(10); 10
```

Agora coloque a palavra chave *explicit* na frente da declaração do construtor.

```
Saída com uso de explicit:
-----
[andre@mercurio Cap3-P00UsandoC++]$ g++ e96-explicit.cpp
e96-explicit.cpp: In function 'int main ()':
e96-explicit.cpp:36: could not convert '10' to 'const TMatriz &'
e96-explicit.cpp:15: in passing argument 1 of 'Print (const TMatriz &)'

Ou seja, com explicit o compilador não aceita a chamada
Print(10);
*/
```

Para evitar a conversão automática na linha `Print(10);` deve-se usar a palavra chave `explicit` na declaração da função construtora, da seguinte forma:

```
Exemplo:
//construtor
explicit TMatriz (int _x = 10);
//Para chamar explicitamente use:
Print ( TMatriz(5) );
```

20.6 Sentenças para construtor e métodos de conversão

- Podemos realizar conversões com construtores e com métodos de conversão.
- Quando temos herança podemos ter a necessidade de construir construtores de conversão.
- Conversões definidas pelo programador somente são usadas implicitamente se forem não ambíguas.
- Seja cauteloso com conversões implícitas.
- Com construtores, não podemos converter um objeto da classe em um outro tipo, pois o construtor não pode ter retorno. Ou seja, com construtores só podemos realizar conversão para tipos da própria hierarquia.
- Um método de conversão em uma classe derivada não oculta um método de conversão em uma classe básica, a menos que os dois métodos convertam para o mesmo tipo.

- Métodos de conversão podem ser virtuais.
- A biblioteca padrão usa métodos de conversão, no exemplo abaixo, o operador while espera um bool (0 ou != 0), enquanto a chamada a cin >> x, for ok, cin é convertido em true, se cin >> x falha, retorna false.

Exemplo:

```
while(con>>x)
cout << "objeto : "<< x << " lido.";
```

- Reduza o número de funções construtoras usando argumentos default.
- Com a palavra chave explicit, o construtor só vai ser chamado de forma explícita.
- Use explicit e evite surpresas.
- ² Para converter da classe C para a classe B (sendo B não herdeira de C), deve-se criar uma função de conversão dentro da classe C, e declarar como friend dentro de B.
- ³ Conversões implícitas e funções friend não podem ser criadas com referências simples (&).

Exemplo:

```
friend ...return x& ; //erro
//mas podem ser criados para
friend...return ...const X& ;
friend....return...X;
```

20.7 Conversão dinâmica com dynamic_cast

Quando temos uma hierarquia de classes, podemos criar um ponteiro para a classe base e fazer com que este ponteiro aponte para um objeto de uma classe derivada (veja capítulo sobre polimorfismo, Cap.16).

Em algumas ocasiões é necessário converter o ponteiro da classe base em um ponteiro de classe derivada.

Isto é solucionado com o uso de cast dinâmico, um cast realizado em tempo de execução e que usa a palavra chave dynamic_cast. Veja o protótipo.

Protótipo:

```
tipoBase* ptrbase;
ptrbase = new tipoDerivado;
tipoDerivado* ptr = dynamic_cast <tipoDerivado*> (ptrbase);
```

Converte o ponteiro ptrbase para um ponteiro do tipo tipoderivado. Funciona se tipoDerivado é derivado direta ou indiretamente do tipobase, caso contrário retorna 0. Veja a seguir um exemplo.

Exemplo:

```
class TMatriz{};
class TImagem:public TMatriz{};
TMatriz* pm;          //ponteiro para classe base
pm = new TImagem(); //aponta para objeto derivado
.....//pm é usado
TImagem * im = dynamic_cast < TImagem* > (pm);
```

Neste exemplo, cria-se um ponteiro pm que aponta para um objeto TImagem (derivado). Posteriormente cria TImagem* a partir de pm, usando o dynamic-cast. Funciona porque o objeto criado e apontado por pm é um objeto do tipo TImagem.

Apresenta-se no exemplo a seguir o uso de dynamic_cast. Neste exemplo inclui-se o uso de excessões, que serão descritas no Capítulo 21.

Listing 20.2: Uso do dynamic-cast.

```
//Cobre uso de cast dinâmico, verificador de tipo, e excessões.
//Extraído do HELP do Borland C++ 5.0 e adaptado
# include <iostream>
# include <typeinfo>

//-----Base1
class Base1
{
    virtual void f() { };
};

//-----Base2
class Base2
{ };

//-----Derivada
class Derivada : public Base1, public Base2
{ };

//-----Main
int main()
{
    try
    {
        Derivada d;          //cria objeto d
        Derivada *pd;       //ponteiro para Derivada
        Base1 *b1 = & d;    //ponteiro para Base1, aponta para objeto d

        cout<<"Realiza um cast dinâmico (downcast) de Base1 para Derivada."<<endl;
        if ((pd = dynamic_cast<Derivada *>(b1)) != 0)
        {
            cout << "Tipo do ponteiro resultante = " << typeid(pd).name() << endl;
        }
    }
    else
        throw bad_cast(); //previamente definido

    // Estar atento a hierarquia da class.
```

```

//isto é, cast de uma classe B1 para D e depois de D para B2
Base2 *b2;

    cout<<"Realiza um cast dinâmico (downcast) de Base1 para Base2."<<endl;
if ((b2 = dynamic_cast<Base2 *>(b1)) != 0)
    {
        cout << "Tipo do ponteiro resultante = " << typeid(b2).name() << endl;
    }
else
    throw bad_cast();
}

catch (bad_cast)
    {
        cout << "0 dynamic_cast falhou" << endl;
        return 1;
    }
catch (...)
    {
        cout << "Excessão... disparada." << endl;
        return 1;
    }
return 0;
}

/*Saída:
-----
[andre@mercurio Cap3-P00UsandoC++]$ ./a.out
Realiza um cast dinâmico (downcast) de Base1 para Derivada.
Tipo do ponteiro resultante = P8Derivada
Realiza um cast dinâmico (downcast) de Base1 para Base2.
Tipo do ponteiro resultante = P5Base2
*/

```

20.7.1 Sentenças para cast dinâmico

- A sigla RTTI significa run-time information, permite o uso de conversões com o `dynamic_cast`.
- Para uso do cast dinâmico é necessária a passagem do parâmetro `-RTTI` para o compilador (leia informações de seu compilador).
- O uso do RTTI e do `dynamic_cast` implica em códigos maiores e mais lentos.
- Sempre testar o ponteiro após o `dynamic_cast`.
- Se a classe destino for privada ou protegida, o `dynamic_cast` falha.
- ²Observe que você não deve usar RTTI com o tipo `void*`.
- ²Prefira `dynamic_cast` a `typeid`.

- ² Se numa hierarquia você tiver classes duplicadas, bases virtuais, acessíveis por mais de um caminho, você deve verificar se o cast desejado não é ambíguo. Se for ambíguo, o `dynamic_cast` retorna 0.
- ³ Veja o template `reinterpret_cast<tipo>(arg)`; visto na seção conversão com `reinterpret_cast`.

20.8 Conversão estática com `static_cast`

O `static_cast` é usado para conversões de tipos em tempo de compilação, daí o termo `static`. Se a conversão for ilegal, o compilador acusa o erro.

Protótipo:

```
tipo_old obj;  
tipo_obj2 = static_cast<tipo>(obj);
```

Exemplo:

```
float fpi = 3.141516;  
double dpi = static_cast<double>(fpi);
```

20.9 Conversão com `reinterpret_cast`

Permite reinterpretar um cast. Não use `reinterpret_cast`, se em algum momento precisar dele é sinal de que sua modelagem está com problemas.

Exemplo:

```
int main()  
{  
    int a = 3;  
    int* pa = &a;  
    cout << "pa=" << pa << endl;  
    cout << *reinterpret_cast<char*>(pa) << endl;  
}
```

20.10 Usando `Typeid`

Para facilitar a verificação do tipo de determinado objeto, foi desenvolvida a biblioteca `<typeinfo>`. A biblioteca `<typeinfo>` sobrecarrega os operadores `==` e `!=` para comparar tipos do usuário.

O operador `typeid` pode ser usado para verificar o tipo de um objeto.

Veja no exemplo abaixo, como comparar dois objetos A e B, para saber se são do mesmo tipo.

Listing 20.3: Uso de `typeid`.

```
#include <iostream>  
#include <string>  
#include <iomanip>  
#include <typeinfo>
```

```

using namespace std;

class A
{
public:
    int a;
};

class B : public A
{
public:
    int b;
};

class K
{
public:
    int k;
};

void main()
{
    A a; //cria objeto do tipo A com nome a
    B b; //cria objeto do tipo B com nome b
    K k; //cria objeto do tipo K com nome k

    cout << "(typeid(a)==typeid(a))_UUUU->" <<(typeid(a)==typeid(a))<<endl;
    cout << "(typeid(a)==typeid(b))_UUUU->" <<(typeid(a)==typeid(b))<<endl;
    cout << "(typeid(a)==typeid(k))_UUUU->" <<(typeid(a)==typeid(k))<<endl;
    cout << "(typeid(b)==typeid(k))_UUUU->" <<(typeid(b)==typeid(k))<<endl;

    cout << "␣typeid(a).name()_UUUUUUUUUU->" << typeid(a).name()<<endl;
    cout << "␣typeid(b).name()_UUUUUUUUUU->" << typeid(b).name()<<endl;
    cout << "␣typeid(k).name()_UUUUUUUUUU->" << typeid(k).name()<<endl;

//nome=int
    int intObject = 3;
    string nomeintObject (typeid(intObject).name());
    cout << "nomeintObjectUUUUUUUUUU->" << nomeintObject <<endl;

//nome= doubleObject
    double doubleObject = 3;
    string nomedoubleObject (typeid(doubleObject).name());
    cout << "nomedoubleObjectUUUUUUUUUU->" << nomedoubleObject <<endl;
}

/*
Novidade:
-----
Uso de typeid para verificar se dois objetos são do mesmo tipo
E para obter o nome de identificação da classe do objeto.
*/

/*
Saída:

```



```

-----
[andre@mercurio Cap3-POOUsandoC++]$ ./a.out
(typeid(a)==typeid(a))    ->1
(typeid(a)==typeid(b))    ->0
(typeid(a)==typeid(k))    ->0
(typeid(b)==typeid(k))    ->0
  typeid(a).name()        ->1A
  typeid(b).name()        ->1B
  typeid(k).name()        ->1K
nomeintObject             ->i
nomedoubleObject          ->d
*/

```

20.11 Verificação do tamanho de um objeto com sizeof

O operador sizeof é utilizado para retornar o tamanho de um objeto. Veja as possibilidades:

- O uso de sizeof(tipo) retorna o tamanho do tipo.
- O uso de sizeof(objeto) retorna o tamanho do objeto.
- O uso de sizeof(ptr), retorna o tamanho da classe do ponteiro.

Exemplo:

```

classeBase *ptr = new classeDerivada;
sizeof(classeBase);    //retorna o tamanho da classe base
sizeof(classeDerivada); //retorna o tamanho da classe Derivada
sizeof(ptr);           //retorna tamanho da classe base

```

20.12 Referências e *dynamic_cast*

O `dynamic_cast` também pode ser utilizado com referências.

Protótipo:

```

tipo& ref = dynamic_cast< Tipo& >(r);

```

Se o `dynamic cast` para uma referência falha, ocorre uma exceção do tipo `bad_cast`.

Capítulo 21

Excessões

Neste capítulo vamos apresentar as excessões. Os conceitos básicos e o uso de excessões. Como fica a sequência de controle em um programa com excessões.

21.1 Introdução as excessões

O uso de excessões permite a construção de programas mais robustos, com maior tolerância a falhas. O exemplo abaixo mostra um problema usual em programas, a divisão por zero.

Listing 21.1: Excessão: Divisão por zero .

```
#include <iostream>
using namespace std;
void main()
{
    //Uma divisão por zero sem controle de erro
    float a = 3.0;
    float b = 0.0;
    float c = a / b;
    float d = c;
    cout << "a=" << a << " b=" << b << " c=a/a/b b="
        << a/b << " d=" << d << endl;
}
/*
Novidade:
Programa com bug, divisão por zero.
No Linux/GNU/g++ aparece c = inf (de infinito)
*/
/*
Saída:
-----
[andre@mercurio Cap3-P00UsandoC++]$ ./a.out
a=3 b=0 c=a/b=inf d=inf
*/
```

A solução usualmente adotada para resolver este problema é algo como:

Listing 21.2: Excessão: Divisão por zero com controle simples.

```
#include <iostream>
```

```

using namespace std;
void main()
{
//Uma divisão por zero com controle de erro
float a = 3.0;
float b = 0.0;
float c = a / b;
cout << "Entre com b:";
cin >> b;
cin.get();
if( b == 0) //controle
    cout << "Erro b=0" << endl;
else
    {
        c = a / b;
        cout << "c=a/b=" << c << endl;
    }
}
/*
Novidade:
Programa com controle simples
*/
/*
Saída:
-----
[andre@mercurio Cap3-P00UsandoC++]$ ./a.out
Entre com b:6
c = a / b=0.5
[andre@mercurio Cap3-P00UsandoC++]$ ./a.out
Entre com b:0
Erro b = 0
*/

```

Observe que embora o programa não tenha mais um bug, existe a necessidade de se criar um conjunto de flags e verificações adicionais.

De uma maneira geral, o controle de erros em um programa pode ser feito de duas formas.

- No meio do código (forma usual), controlando a entrada de dados e os fluxos do programa. Esta metodologia de controle deixa o código mais confuso, pois o tratamento dos erros fica no meio do código.
- Usando as técnicas de tratamento de excessões, que permitem uma formulação mais robusta para o tratamento e verificação de erros em um programa.

21.2 Conceitos básicos de excessões

Uma exceção é uma condição excepcional que ocorre em um programa o que exige um tratamento especial, é composta basicamente de três elementos:

- Um bloco *try* que contém todo o código de programa a ser executado.
- Uma ou mais chamadas a *throw*, *throw* lança uma exceção, ou seja, anuncia que ocorreu um erro. O *throw* lança um objeto que será capturado por um bloco *catch*.

- Um ou mais blocos *catch*, que são responsáveis pelo tratamento das exceções lançadas com *throw*.

O exemplo abaixo ilustra o processo:

Listing 21.3: Excessão: Divisão por zero com excessões.

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

void main()
{
    //Uma divisão por zero com tratamento de excessões
    float a = 3.0;
    float b ;
    float c ;
    try
    {
        cout << "Entre com b:";
        cin >> b;
        cin.get();

        if( b == 0) throw string("Divisão por zero");//out_of_range;
        c = a / b ;
        cout<< "c=a/b=" << c << endl;
    }

    catch(string msg)
    {
        cout << "Excessão:" <<msg<< endl;
    }
}

/*
Novidade:
Uso de tratamento de excessões.
Se b=0, lança uma excessão, lança um objeto string que é capturado pelo catch.
*/
/*
Saída:
-----
[andre@mercurio Cap3-P00UsandoC++]$ ./a.out
Entre com b:1
c = a / b = 3
[andre@mercurio Cap3-P00UsandoC++]$ ./a.out
Entre com b:0
Excessão: Divisão por zero

*/
```

Observe que uma excessão permite a um método ter um retorno diferente do especificado, alterando a linha de execução do programa.

Descreve-se a seguir os três componentes de uma excessão.

21.2.1 try

Os blocos try podem lançar uma ou mais exceções, lançando diferentes tipos de objetos.

Dentro do bloco try você pode lançar diretamente uma exceção com throw ou chamar métodos (que podem ser encadeados), e que em algum momento lancem uma exceção com throw.

Exemplo:

```
try {f1();};           //tenta executar o bloco
f1(){f2();};         //f1 chama f2
f2(){f3();};         //f2 chama f3
f3(){throw "erro";}; //f3 lança uma exceção
```

O lançamento de uma exceção por um método termina imediatamente a execução do método, e chama o próximo bloco catch que receba o tipo de objeto lançado.

21.2.2 throw

O throw é usado para lançar a exceção, é como uma chamada de um método. O throw funciona como um retorno multi-nível, visto que o retorno do método é diferente do especificado.

Se um método lança uma exceção com throw, você pode mudar sua declaração, incluindo as exceções que podem ser lançadas.

Exemplo:

```
//Declaração de método informando a exceção que pode ser lançada
void funcao1() throw (objeto& ob);
//Declaração de método que pode lançar mais de um tipo de exceção
void funcao2() throw (objeto& ob, string s);
```

- A vantagem em se declarar os tipos de exceções que um método pode lançar, é que o compilador confere, e impede que o método lance uma exceção não declarada em seu protótipo.

21.2.3 catch

O catch é o tratador de exceções. Logo após o bloco try você deve ter o bloco catch.

Exemplo:

```
//Trata string
catch(string s) {.....};
//Trata int
catch(int i) {.....};
//Trata qualquer tipo de exceção, deve ser o último catch
catch(...) {.....};
```

- Como um bloco try pode lançar mais de um tipo de exceção, você pode ter mais de um bloco catch, um para cada objeto lançado.
- Observe que o bloco catch só é executado se tiver sido lançada uma exceção com throw.

- Se foi lançado um objeto do tipo X, o bloco catch a ser executado é aquele que trata um objeto do tipo X.
- Um grupo de tratamento catch se assemelha a um if else encadeado.
- Como um catch(...) trata qualquer tipo de exceção, o mesmo deve ser o último cast. Observe ainda que como catch(...) não recebe nenhum objeto não pode fazer muita coisa.

21.3 Sequência de controle

A sequência de controle do programa pode ser a normal, isto é, sem ocorrência de exceções e a com ocorrência de exceção.

21.3.1 Sequência de controle sem exceção

Quando não ocorre nenhuma exceção, a sequência executada é dada pelos números 1,2,3,4,5.

```
Exemplo:  
try  
{          //1  
....      //2  
throw B; //não executado  
...       //3  
}         //4  
catch (A)  
{...}  
catch (B)  
{...}  
catch (C)  
{...}  
restante do código... //5
```

Observe que como não ocorre a exceção, isto é, a linha **throw B**; não é executada. Depois do bloco try, executa a linha abaixo do último bloco catch, ou seja, executa a sequência 1,2,3,4,5.

21.3.2 Sequência de controle com exceção

No caso em que ocorre uma exceção a sequência de controle é modificada, veja o exemplo. A sequência é dada pelos números 1,2,3,4,5,6.

```
Exemplo:  
try  
{          //1  
....      //2  
throw B; //3 (executado)  
...  
}
```

```

catch (A)
{
...
}
catch (B) //4
{...} //5
catch (C)
{...}
restante do código... //6

```

Observe que como ocorre a excessão do tipo B, isto é, a linha **throw B**; é executada. O bloco try é encerrado na linha em que ocorre o throw, a seguir é executado o bloco catch (B) e depois a sequência continua em restante do código. Isto é, executa a sequência 1,2,3,4,5,6.

21.4 Como fica a pilha (heap)²

Quando uma excessão é lançada, os objetos locais e os objetos dinâmicos alocados com `auto_ptr` dentro do bloco try são destruídos. A seguir, o próximo catch que trata a excessão lançada é executado. Se este catch não existe no escopo do try que lançou a excessão, todas as demais chamadas de métodos que estão penduradas na pilha e que foram incluídas neste bloco try são desempilhadas (desconsideradas).

Releia o parágrafo anterior e veja que faz sentido, pois se houve problema no bloco try, suas sub-rotinas devem ser descartadas. Veja o exemplo.

Listing 21.4: Excessão e desempilhamento.

```

//-----Inclusão de arquivos
#include <iostream>
#include <new>
#include <vector>
#include <stdexcept>
#include <string>

using namespace std;

//-----class Teste
class Teste
{
public:
void f3(int resp)
{
    cout << "Início_f3."<<endl;
    if(resp==1)
        throw (string("Funcao_3"));
    cout << "Fim_f3."<<endl;
}

void f2(int resp)
{
    cout << "Início_f2"<<endl;
    f3(resp);
}
}

```



```

    cout << "Fim_f3."<<endl;
}
void f1(int resp)
{
    cout << "Início_f1."<<endl;
    f2(resp);
    cout << "Fim_f1."<<endl;
}
};

//-----Função main()
int main()
{
    int resp;
    cout<<"\nDeseja_executar_sem_excessão(0)_ou_com_excessão(1):";
    cin >> resp; cin.get();

    Teste obj;
try
{
    obj.f1(resp);
}

catch(string s)
{
    cout <<"\n0correu_Excessao_na_função:"<<s<<endl;
}
}

/*
Novidade:
-----
Uso de excessões.
Verificação do desempilhamento das chamadas dos métodos em um bloco
try que sofreu a chamada a throw.
*/

/*
Saída:
=====
[andre@mercurio Cap3-P00UsandoC++]$ ./a.out

Deseja executar sem excessão (0) ou com excessão (1):0
Início f1.
Início f2
Início f3.
Fim f3.
Fim f3.
Fim f1.

[andre@mercurio Cap3-P00UsandoC++]$ ./a.out
Deseja executar sem excessão (0) ou com excessão (1):1
Início f1.
Início f2
Início f3.

```

```
Ocorreu Excessao na função : Funcao 3
*/
```

21.5 Excessões não tratadas

As exceções não tratadas terminam a execução do programa.

Quando uma exceção é lançada e não capturada, as seguintes funções são executadas: *unexpected*, que chama *terminate*, que chama *abort*, que finaliza o programa.

Você pode criar suas próprias funções *unexpected* e *terminate*; e substituir as default com as funções *set_terminate* e *set_unexpected*.

Exemplo:

```
//Cria função sem retorno e sem parâmetro
void minhaFuncaoTerminate(){};
//Define como terminate
set_terminate(minhaFuncaoTerminate);
```

21.6 Excessão para new

De uma maneira geral *new* tenta alocar memória, se falhar lança uma exceção, se a mesma não tiver tratamento chama *abort* ou *exit* da `<cstdlib>`.

Nos novos compiladores (Ansi C++) quando *new* falha, em vez de retornar 0, retorna um `throw(bad_alloc)`.

Veja a seguir um exemplo.

Listing 21.5: Excessão para new.

```
#include <iostream>
#include <new>
#include <vector>
struct S
{
    int indicador[5000];
    double valor [5000];
};

int main()
{
    vector<S> v;
    //Uso de try
    try
    { long int i=0;
      //um for infinito
      for (;;)
      {
          S * ptr_s = new S();
          v.push_back(*ptr_s);
          cout<<"\nv["<<i<<"]_alocada"<<endl;
          i++;
      }
    }
```

```

    }
catch (bad_alloc erro)
{
    cout << "\nExcessao_" << erro.what() << endl;
}
}

/*
Novidade:
Uso de excessões para new.
O programa cria uma estrutura. A seguir, dentro de main, cria um vetor.
No bloco do for infinito, aloca uma estrutura dinamicamente e acrescenta ao
vetor.
Quando a memória de seu micro terminar, ocorre uma excessão do tipo bad_alloc,
e o bloco catch (bad_alloc é executado.
*/
/*
Saída:
-----
v[8190] alocada

v[8191] alocada
out of memory
*/

```

Se quiser desativar a chamada da excessão para new use nothrow:

Exemplo:

```
ptr[i] = new (nothrow) double[5000];
```

- Você pode criar uma função para tratar a alocação com new, a mesma deve receber e retornar um void e pode ser ativada com set_new_handler. Como você redefiniu o tratamento para new, a excessão para new deixa de ser lançada e sua função é executada.

21.7 Excessões padrões

Apresenta-se a seguir algumas excessões padrões do C++, as mesmas estão listadas nos arquivos <exception> e <stdexcept>.

Excessões:

- bad_alloc //Falha alocação
- bad_cast //Falha conversão
- bad_typeid //Falha verificação de tipo
- bad_exception / Falha de excessão

Erros lógicos:

- invalid_argument //argumento inválido

- `length_error` //dimensão errada
- `out_of_range` //fora do intervalo

Erros de runtime

- `overflow_error` //número muito grande
- `underflow_error` //número muito pequeno

21.8 Sentenças para excessões

- Você pode lançar uma exceção dentro de um construtor, se a mesma ocorrer os demais objetos deixam de ser criados, deixando de ser destruídos.
- Uma exceção não resolve os destinos do seu programa, esta tarefa ainda é sua.
- Só use exceção para casos críticos, evite seu uso como mais um mecanismo de programação.
- Os métodos para tratamento de excessões e aqueles de tomada de decisão devem ser separados do código numérico.
- Observe que em um sistema tradicional de tratamento de erros, as instruções de tratamento de erro ficam misturadas no código. Com o uso de `try`, `throw` e `catch` você separa a região do código da região de tratamento de erros.
- Veja no capítulo BUG o uso da instrução `assert`.
- Disparar uma excessão com `throw` fora de um bloco `try` provoca o encerramento do programa, executando `terminate`.
- Dentro de um método você pode declarar objetos locais; que serão eliminados quando do encerramento do método. Mas se dentro do método for lançada uma exceção com `throw`, o método não termina e os objetos locais não são eliminados. Para resolver este problema, deve-se ativar a opção `Options->project->RTTI` do compilador, que assegura que os objetos locais tenham seus destrutores chamados.
- Mesmo ativando a opção RTTI, somente os objetos locais que não são dinâmicos são destruídos. Objetos dinâmicos devem ser encerrados antes do lançamento da exceção, pelo programador.
- TRY
 - Depois de um bloco `try` ou `catch` não coloque o `;` (ponto e virgula).
- THROW
 - Uma excessão lançada com `throw` é considerada tratada (encerrada), assim que entra no `catch`.

- *CATCH*

- A sequência dos catches deve respeitar a hierarquia das classes. Preste atenção no exemplo, o cast da classe derivada deve vir antes do da classe base.

Exemplo:

```
//Errado
catch(classe Base)...
catch(classe Derivada1)...
catch(classe Derivada2)...
//Correto
catch(classe Derivada2)...
catch(classe Derivada1)...
catch(classe Base)...
```

- Se você dispara com um throw um ponteiro ou referência de um objeto do tipo A, a mesma deve ser capturada por um catch que recebe um ponteiro ou referência do tipo A. Daí você conclue que um catch (void*) captura todos os lançamentos de ponteiros e deve ser o último deste tipo.
 - Se throw lança um const, catch deve capturar um const.
- Leia os exemplos apresentados com atenção, e veja que o uso de excessões é simples.

Capítulo 22

Templates ou Gabaritos

Os conceitos mais úteis e extraordinários de C++ são o conceito de classes, de polimorfismo e de templates. Apresenta-se neste capítulo o conceito de template (gabaritos).

22.1 Introdução aos templates (gabaritos)

Uma função template é uma função genérica em que o tipo dos dados (parâmetros e de retorno) são definidos em tempo de compilação.

No capítulo sobrecarga de métodos verificamos que a sobrecarga de métodos permite criar métodos homônimos para manipular diferentes tipos de dados:

Exemplo:

```
float f(float x)    {return(x*x);}
int    f(int x)     {return(x*x);}
double f(double x) {return(x*x);}
```

Observe como as funções são semelhantes, a única diferença é o tipo do parâmetro e o tipo do retorno que esta sendo tratado, primeiro um float depois um int e um double.

O ideal seria escrever uma função genérica da forma

```
tipo f(tipo x) {return (x*x);}
```

funções template de C++ funcionam exatamente assim.

Uma função template é uma função genérica, definida para um tipo genérico.

O compilador gera uma cópia da função para cada tipo para a qual ela venha a ser chamada, isto é, se a função for chamada tendo como parâmetro um int cria uma função para int, se o parâmetro for double cria uma função para double, e assim por diante.

22.2 Protótipo para templates

Apresenta-se a seguir o protótipo para funções templates.

Protótipo da função template:

```
//Formato 1
```

```

template <class tipo_1,...,class tipo_n>
tipo_i nome_função(tipo_i nome)
{//definição_da_função_template};
//
//Formato 2
template<typename tipo_1,...,typename tipo_n>
tipo_i nome_função(tipo_i nome)
{};

```

Exemplo:

```

template <class tipo1>
tipo1 f(tipo1 a) {return(a*a);}

```

Assim, a função template é uma função que vai receber um objeto do tipo_1 e multiplicá-los. Retornando o resultado da multiplicação, que é um objeto do tipo1.

```

Exemplo;
template <typename T1>
void Print(const T1* matriz, const int size)
{
for (int i = 0; i < size; i++)
cout << matriz[i] << endl;
}

```

No exemplo acima, podem ser criadas funções como:

```

void Print(const int* matriz, const int size);
void Print(const float*matriz, const int size);
void Print(const char* matriz, const int size);

```

No exemplo acima vai criar a função Print para int, depois para float e depois para char.

Resumo:

Tudo bem, você ainda não entendeu como funciona o template. Bem, vamos lá, template pode ser traduzido para o português como gabarito.

É traduzido como gabarito mas funciona como um gabarito ? Exatamente.

E como C++ implementa estes gabaritos ?

É simples, é como um copy/past inteligente.

Ao encontrar uma declaração de uma função template, o compilador verifica se a declaração esta ok. Se a mesma estiver ok, o compilador armazena a declaração num bufer (na memória).

Quando você chama a função declarada com o template, passando como parâmetro um int, o compilador do C++ reescreve toda a função, substituindo o Tipo por int. O compilador do C++ faz um search (Tipo) replace (int). Esta busca e substituição é repetida para cada tipo.

22.2.1 Declaração explícita de função template

Como visto, a função template só será implementada para inteiros quando o compilador encontrar uma chamada da função que use inteiros.

Se você deseja que uma função template, tenha uma versão para inteiros, basta declarar explicitamente a função para inteiros:

```
Exemplo:  
int função_template(int, int);
```

Observe que você esta apenas declarando a função, sua definição será dada automaticamente pelo compilador, tendo como base a definição do template.

22.2.2 Sobrecarga de função template

Se você deseja que uma função tenha o mesmo nome de uma função definida como template, mas que receba um número de parâmetros diferente, não tem problema, você esta sobrecarregando a função template. Observe entretanto que o número de parâmetros de entrada desta função deve ser diferente dos definidos para a função template, senão o compilador vai acusar ambiguidade.

```
Exemplo:  
tipo f(tipo a, tipo b);  
tipo f(tipo a, tipo b, tipo c);
```

22.2.3 Função template com objeto estático

Dentro de uma função template você pode definir um tipo1 como objeto estático.

Ao criar uma função para inteiros, cria um inteiro estático. Ao criar para float, cria um float estático. Estes objetos não se interferem pois estão em funções distintas.

22.3 Classes templates (ou tipos paramétricos)

Uma classe template implementa o conceito de uma classe genérica. Uma classe que pode ser construída para mais de um tipo, mas que tem a mesma forma (estrutura).

Veja exemplo de classe template no capítulo “class <complex>”, onde apresenta-se a classe <complex>. A classe <complex> representa números complexos. A classe pode ser construída para números float, double e long double.

22.4 Sentenças para templates

- Desaconselha-se o uso de funções globais, o mesmo é válido para funções template globais.
- Não crie templates antes de ter total confiança no código, isto é, primeiro crie classes normais e faça todos os testes que for necessário, depois pode implementar sobre esta classe testada a classe templatizada.

- Quando o relacionamento das classes se torna complexo, o compilador pode ter problemas de resolução de acesso.

Agora o leitor pode compreender as vantagens da programação orientada a objeto. O uso de conceitos abstratos como classes, que permitem a criação de tipos do usuário, adequados a solução de seus problemas; do polimorfismo, que permite que o código fique altamente genérico e pequeno; da sobrecarga de operadores, que permite ao programador definir operações conhecidas como `++`, `*/`, e o uso de templates que tornam o código genérico. Fazendo de C++ uma linguagem de programação extremamente poderosa.

Parte III

Classes Quase STL

Capítulo 23

Entrada e Saída com C++

Apresenta-se neste capítulo as classes fornecidas pelo C++ para entrada e saída de dados.

23.1 Introdução a entrada e saída de dados no c++

Nos capítulos anteriores e nos exemplos apresentados você aprendeu a usar `cin` e `cout` para entrada e saída de dados.

```
Exemplo:
using std;
//Cria objeto do tipo int com nome x
int x;
//Enviar para tela o conjunto de caracteres
cout <<"Entre com o valor de x" ;
//Pegar os caracteres digitados no teclado e armazenar em x
cin >> x;
cin.get();
//Enviar para tela o valor de x
cout << "valor de x= " << x << endl;
```

O aspecto fundamental a ser entendido é que a entrada e saída de dados em C++ funciona com um fluxo de caracteres de um objeto para outro, e que os operadores utilizados para enviar estes caracteres de um objeto para o outro, são os operadores de inserção (<<) e de extração (>>).

No capítulo de Tipos, você aprendeu que programar em C++ se traduz em conhecer a sintaxe de C++ e a usar 3 tipos de objetos. Os tipos básicos de C++ (`char`, `int`, `float`, `double`,...), os tipos definidos pelo usuário e os tipos definidos em bibliotecas externas (como a STL).

Pois bem, este capítulo apresenta uma hierarquia de classes oferecidas pelo C++ e que possibilitam a formatação e a manipulação avançada da entrada e saída de dados.

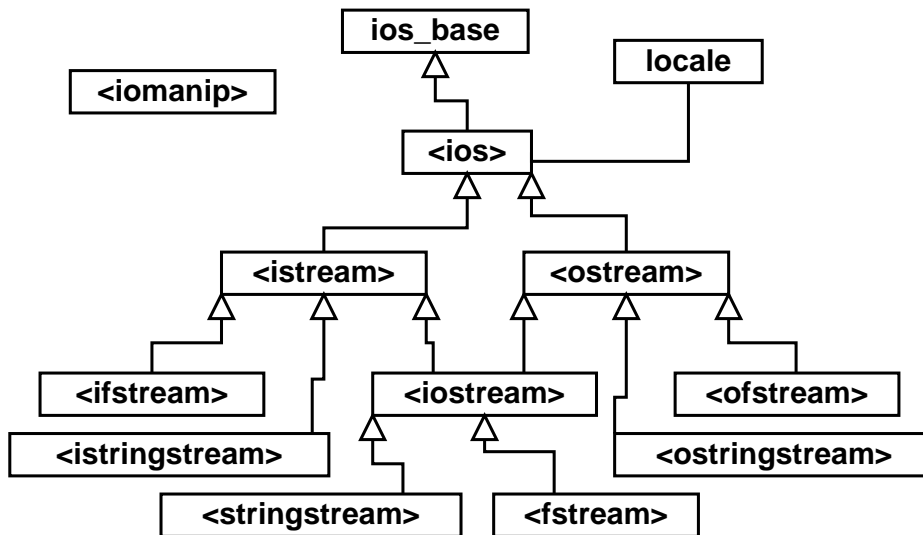
23.1.1 Biblioteca de entrada e saída

Um objeto da hierarquia de entrada e saída é uma stream. Um objeto stream contém um buffer onde ficam armazenados os caracteres, de entrada ou saída, e um conjunto de atributos utilizados na formatação destes caracteres.

A Figura 23.1 ilustra a hierarquia desta biblioteca. Um objeto desta hierarquia pode armazenar, receber e fornecer caracteres.

Esta biblioteca é templatizada, isto é, é construída de forma genérica, e pode trabalhar com caracteres comuns (char) ou caracteres estendidos (wchar)¹.

Figura 23.1: Esboço da biblioteca de manipulação de entrada e saída.



A classe `<ios_base>`² contém um conjunto de métodos básicos que são herdados pelas demais classes da hierarquia. A classe `<ios_base>` contém a função `setf` e um conjunto de atributos que podem ser utilizados para definição do formato de saída de dados. A classe `<iomanip>` contém um conjunto de manipuladores que podem ser utilizados para formatação da saída de dados. A classe `<istream>` é usada para entrada de dados, C++ fornece automaticamente o objeto `cin`, do tipo `istream` para leitura de dados do teclado. A classe `<ostream>` é usada para saída de dados, C++ fornece automaticamente o objeto `cout` do tipo `ostream`, usado para enviar a saída para a tela.

Em alguns casos deseja-se enviar e receber dados para um arquivo de disco, nestes casos usa-se a classe `<fstream>` e as associadas `<ofstream>` e `<ifstream>`. A classe `<stringstream>` é uma mistura da classe `<iostream>` com a classe `<string>`, funcionando, ora como uma string ora como uma `iostream`.

A hierarquia de classes ilustrada na Figura 23.1 foi desenvolvida utilizando o conceito de templates (gabaritos). Isto significa que a mesma foi implementada para tipos genéricos de caracteres. Na prática as classes são construídas para dois tipos de caracteres. O `char`, já usado no C, e o `wchar`. O `char` suporta 255 caracteres e o `wchar` cerca de 16000 caracteres. O `wchar` foi desenvolvido para dar suporte a diversas linguagens (inglês, português, ...).

Antes de iniciarmos a descrição de cada classe da biblioteca de entrada e saída de dados, vamos descrever, brevemente o `locale`.

¹Um `char` tem 255 caracteres, um `wchar` tem ~16000 caracteres.

²Estruturas de `<ios_base>` estão definidas em `<ios>`, `basic_istream` em `<istream>`, `basic_ostream` em `<ostream>`. Exceto `<ios_base>`, todas as classes da hierarquia são gabaritos, cujo nome inicia com `basic_`. Na descrição destas classes, tirei o `basic_`.

23.2 O que é um locale ?

Como dito, as classes foram desenvolvidas utilizando gabaritos e de uma forma genérica. Foi previsto um sistema denominado de locale. Um locale é um conjunto de definições que permitem que você escreva programas em múltiplas linguagens.

Um locale é um objeto de formatação que especifica a forma como os caracteres utilizados serão tratados, considerando características relativas a diferentes linguagens escritas. Isto é, um locale define a forma como os caracteres serão tratados.

Para detalhes do funcionamento de um locale, consulte o livro [?] (3 edição revisada em Inglês). Infelizmente, a edição traduzida para o Português, a terceira, não contempla o apêndice sobre locale. Entretanto, o autor disponibilizou no seu site (<http://www.research.att.com/~bs/3rd.html>), o capítulo sobre locale³ (em Inglês), você pode baixar e imprimir.

23.3 A classe <ios_base>

A classe `ios_base` contém informações de formatação da stream. Não considera informações do locale.

Métodos de <ios_base>

Formatação da stream

int setf();

O método `setf` é usado para definir diversos atributos da stream que são usados na formatação de saída (veja na Tabela 23.1 alguns destes atributos).

int unsetf();

Desativa formatação de saída.

int rdstate();

Retorna o estado atual do fluxo.

Um dos principais métodos da classe <ios_base> é o método `setf()`, que é usado com o objetivo de definir a formatação de saída. Veja na Tabela 23.1 os flags que podem ser usados para alterar os atributos da stream através do método `setf`. Veja a seguir um exemplo.

Exemplo:

```
using namespace std;
//ativa notação científica
cout.setf(ios::scientific);
//desativa notação científica
cout.unsetf(ios::scientific);
//alinhamento a esquerda
```

³Também estão disponíveis no mesmo site: A Tour of C++ presenting the basic programming techniques supported by C++ and the language features through which C++ supports them. A Tour of the Standard Library presenting a few basic uses of C++ introducing its standard library; for most people this chapter gives a better view of C++ than does "A Tour of C++" and Appendix D: Locales presenting C++'s facilities for internationalization.

Tabela 23.1: Flags para o método setf.

Flag de ios_base	Significado
ios_base::skipws	Ignora espaços em branco(somente entrada)
ios_base::left	Alinhamento a esquerda
ios_base::right	Alinhamento a direita
ios_base::internal	Coloca o caracter de preenchimento entre o sinal +/- e o número.
ios_base::shombase	Mostra indicador de base (só saída)
ios_base::showpoint	Mostra ponto decimal(pto flutuante) zeros não significativos no final
ios_base::uppercase	Maiúscula para saída
ios_base::showpos	Mostra sinal + se maior que 0
ios_base::scientific	Usa notação científica
ios_base::fixed	Usa notação fixa
ios_base::unitbuf	Descarrega stream após inserção
ios_base::stdio	Descarrega stdout e stderr após inserção
ios_base::dec	Base 10, d
ios_base::oct	Base 8, o
ios_base::hex	Base 16, h
ios_base::adjustfield	
ios_base::floatfield	

```
cout.setf(ios::left);
//obtém flags
long formatoOriginal = cout.flags();
```

23.4 A classe <ios>

A classe <ios> contém informações de formatação da stream considerando as informações do locale. Isto é, a classe <basic_ios> leva em conta o locale selecionado.

A classe <ios> é uma classe herdeira da <ios_base> e contém funções básicas que podem ser chamadas.

Descreve-se abaixo os métodos de <ios>.

Métodos de <ios>

Estado da stream:

Métodos utilizados para verificar o estado da stream.

int bad();

Verifica se ocorreu um erro nas operações de entrada/saída de dados.
Retorna 0 se ocorreu um erro.

void clear(int = 0);

Zera o fluxo de dados para o estado ok.
Usado para reestabelecer o estado do stream para o estado ok.

int eof();

Verifica se estamos no final do arquivo, observe que eof = end of file.
Retorna um valor diferente de zero no final do arquivo.

int fail();

Igual a zero (= 0) se o estado esta ok, $\neq 0$ se tem erro.

int good();

Igual a zero (= 0) se tem erro, $\neq 0$ se ok.

operator void !();

Diferentes de zero ($\neq 0$) se o estado do fluxo falhou.

Dica: Se estiver bad os caracteres da stream estão perdidos e você deve resetar a stream usando clear(). Se tiver fail, indica que a última operação falhou, mas o stream ainda pode ser utilizado.

Resumo:

- Para saber se esta no final do arquivo
cin.eof();
- Os caracteres foram lidos, mas o formato é incorreto
cin.fail();
- Para saber se houve perda de dados.
cin.badbit();
- Se tudo ocorreu corretamente.
cin.good();
- Para resetar os flags para o estado ok.
cin.clear();

Formatação da stream:

Métodos utilizados para formatação da stream.

width(n);

Seta a largura do campo⁴. Válido apenas para próxima saída.
width seta a largura mínima do campo, isto significa que se a largura é 6 e o número tem 9 dígitos de precisão, vai imprimir os 9 dígitos.

width();

Retorna a largura do campo.

fill(ch);

Seta o caracter de preenchimento.

⁴Um campo com largura n é usada para saída de até n caracteres.

fill():

Retorna o caracter de preenchimento.

precision(int);

Define a precisão da saída, o valor default é 6 caracteres.

precision();

Obtém o valor da precisão da saída.

Veja no exemplo a seguir um exemplo de uso de saída formatada.

Listing 23.1: Formatação básica da saída de dados.

```
//Arquivo ex-Entrada-Saida1.cpp
#include <iostream>
void main()
{
int i = 1;
double d = 1.12345678901234567890;
char c = 'c' ;

//Definindo a largura do campo
cout.width(5);
cout << i << endl;

//Definindo a precisão da saída
for(int i = 1 ; i < 20 ; i++)
{
cout <<"Precisão_";
cout.width(2);
cout << i ;
cout.precision(i);
cout <<"_d="<< d << endl;
}

//Definindo o caracter de preenchimento
cout.fill('*');
cout.width(10);
cout << i << endl;
cout << d << endl;
cout << c << endl;
}

/*
Novidade:
-----
Uso de formatação de saída no C++ (width, fill, precision)

Observações:
-----
Primeiro define o campo em 5 caracteres e imprime o valor i.
Veja que o número 1 é impresso na 5 coluna.
A seguir entra num for onde define a precisão de saída, e imprime o valor de d.
Observe que d é truncado e não arredondado.
No final define o caracter de preenchimento '*' e imprime novamente i.
*/
```

```

/*
Saída:
-----
[andre@mercurio Cap3-POOUsandoC++]$ ./a.out
1
Precisão = 1 d=1
Precisão = 2 d=1.1
Precisão = 3 d=1.12
Precisão = 4 d=1.123
Precisão = 5 d=1.1235
Precisão = 6 d=1.12346
Precisão = 7 d=1.123457
Precisão = 8 d=1.1234568
Precisão = 9 d=1.12345679
Precisão =10 d=1.123456789
Precisão =11 d=1.123456789
Precisão =12 d=1.12345678901
Precisão =13 d=1.123456789012
Precisão =14 d=1.1234567890123
Precisão =15 d=1.12345678901235
Precisão =16 d=1.123456789012346
Precisão =17 d=1.1234567890123457
Precisão =18 d=1.12345678901234569
Precisão =19 d=1.123456789012345691
*****1
1.123456789012345691
c
*/

```

23.5 A classe <iomanip>

A <iomanip> é uma classe de manipuladores. A <iomanip> permite a definição de um conjunto de parâmetros relacionados a formatação da saída de dados. A Tabela 23.2 mostra alguns manipuladores de <iomanip> e o seu significado.

Muitos dos manipuladores de <iomanip> são equivalentes a alguns flags da função `setf` e a métodos da classe <ios_base>, veja a lista a seguir:

- `boolalpha`, `noboolalpha`, `showbase`, `noshowbase`, `showpoint`, `noshowpoint`, `showpos`, `noshowpos`, `skipws`, `noskipws`, `uppercase`, `nouppercase`, `internal`, `left`, `right`, `dec`, `hex`, `oct`, `fixed`, `scientific`, `endl`, `ends`, `flush`, `ws`.

Veja no exemplo o uso de saída formatada usando os manipuladores de <iomanip>. Compare este exemplo com o anterior.

Listing 23.2: Formatação da saída de dados usando `iomanip`.

```

//Arquivo ex-Entrada-Saida2-iomanip.cpp

#include <iostream>
#include <iomanip>
void main()

```

Tabela 23.2: Manipuladores da <iomanip>.

Manipulador	i/o	Significado
endl	o	Insera nova linha "\n" e libera stream (flush)
ends	o	Envia caracter nulo "\0"
flush	o	Esvasia bufer stream
resetiosflags(long fl)	i/o	Define os bits de ios indicados em long fl
setbase(int i)	o	Formata numeros em base i
setfill(char)	i/o	Caracter de preenchimento
setiosflags(lonf fl)	i/o	Define os bits de ios indicados em long fl
setprecision(int pr)	i/o	Define a precisão float para n casas
setw(int w)	i/o	Define o tamanho de campos para w espaços
ws	i	Ignora caracteres em branco
dec	i/o	Formata numeros base 10
oct		Base octal
hex		Base hexadecimal

```

{
int i = 16;
double d = 1.12345678901234567890;
char c = 'c' ;

//Definindo a largura do campo
cout << setw(5) << i << endl;

//Definindo a precisão da saída
for(int cont = 1 ; cont < 20 ; cont++)
{
    cout <<"Precisão_" << setw(2) << cont << setprecision(cont)<<"_d=" << d
        << endl;
}

//Definindo o caracter de preenchimento
cout << setw(10) << setfill('*') << i << endl;
cout << setw(10) << d << endl;
cout << setw(10) << c << endl;

//Definindo formato do número
cout << hex << 15 << endl;
cout << oct << 15 << endl;
cout << dec << 15 << endl;
cout << setbase(10) << 15 << endl;
}

/*
Novidade:
-----
Uso de formatação de saída no C++
*/

```

```

/*
Saída:
-----
[andre@mercurio Cap3-POOU usando C++]$ ./a.out
16
Precisão = 1 d=1
Precisão = 2 d=1.1
Precisão = 3 d=1.12
Precisão = 4 d=1.123
Precisão = 5 d=1.1235
Precisão = 6 d=1.12346
Precisão = 7 d=1.123457
Precisão = 8 d=1.1234568
Precisão = 9 d=1.12345679
Precisão =10 d=1.123456789
Precisão =11 d=1.123456789
Precisão =12 d=1.12345678901
Precisão =13 d=1.123456789012
Precisão =14 d=1.1234567890123
Precisão =15 d=1.12345678901235
Precisão =16 d=1.123456789012346
Precisão =17 d=1.1234567890123457
Precisão =18 d=1.12345678901234569
Precisão =19 d=1.123456789012345691
*****16
1.123456789012345691
c
*****f
17
15
15
*/

```

Dica: Na saída de dados os valores não são arredondados, os mesmos são truncados.

23.6 A classe <istream>

A classe <istream> é uma classe para entrada de dados⁵. Nos diversos exemplos apresentados na apostila, você aprendeu a utilizar o objeto `cin` que é um objeto do tipo <istream>, utilizado para leitura de dados do teclado. Apresenta-se aqui alguns métodos fornecidos pela classe <istream>.

Métodos de <istream>

int gcount();

Retorna o número de caracteres extraídos na última operação de leitura.

istream& ignore(streamsize n=1, int delim=EOF);

Ignora n caracteres, ou até encontrar o delimitador (EOF).

⁵A classe `istream` é uma especialização do gabarito `basic_istream` para caracteres do tipo `char`. Para caracteres do tipo `wchar` use a `wistream`.

int peek();

Retorna o próximo caracter do fluxo sem extrair o mesmo da fila.

istream& putback(ch);

Devolve o caracter ch ao fluxo de entrada. A próxima letra a ser lida será ch.

Exemplo:

```
char ch = k;
cin.putback(ch);
//é como se você tivesse digitado o caracter ch no teclado
```

istream& read(char *s, streamsize n);

Retira n caracteres do array s.

²Desconsidera terminadores e não inclui o caracter de terminação.

long tellg();

Retorna a posição atual do fluxo (posição do ponteiro get).

get();

Obtém um único caracter do teclado, isto é, retira um caracter do teclado.

Exemplo:

```
cin.get();
```

get(ch);

cin.get(ch) obtém um caracter do teclado e armazena em ch. Observe que se o usuário digitou enter a é armazenado em ch e não pega o retorno de carro (o enter digitado pelo usuário). Isto significa que você precisa de um cin.get() adicional para capturar o enter.

Exemplo:

```
char ch, novaLinha;
cin.get(ch);
cin.get(novaLinha); //Captura o enter e armazena em novaLinha
```

get(char*cstring, streamsize n, char='*');

Obtém do teclado até n caracteres, ou até a digitação do asterisco='*',⁶ ou até a digitação do retorno de carro('\n'), o que ocorrer primeiro. O conjunto de caracteres lidos são armazenados em cstring. A função get lê até o terminador, não o incluindo, para pegar o terminador use um cin.get() adicional.

Exemplo:

```
char ch;
cin.get(nomeString, 20);
cin.get(nomeString, 20, 'c');
```

⁶O asterisco representa aqui o caracter terminador, pode ser qualquer caracter.

getline(signed char* cstring, int n, char='\n');

Usada para armazenar uma linha inteira digitado no teclado em uma cstring (string de C). A função getline pega também o retorno de carro. Observe que lê até n-1 caracteres, visto que o último caracter é usado para armazenar o terminador ('\0').

Exemplo:

```
char nome[255];
cin.getline(nome,15);
```

getline(stream cin, string nome, char='\n');

Usada para armazenar uma linha inteira digitado no teclado em uma string de C++. Observe que você inclui dentro de getline a stream.

Exemplo:

```
string nome;
getline(cin, nome);
```

operator >>

Usada para armazenar uma entrada do teclado em uma variável, cin>> lê até o retorno de carro ou até o primeiro espaço em branco (nova linha, tabulador, avanço de formulário, retorno de carro)⁷.

Exemplo:

```
cin >> variável;
cin >> x >> y >> z;
cin >> oct >> numeroOctal;
cin >> hex >> numeroHexadecimal;
```

ignore(int n=1); .

Usada para ignorar até n caracteres do fluxo de caracteres, ou seja, joga fora os próximos n caracteres. Por default ignora um caracter.

Exemplo:

```
cin.ignore(); //ignora 1 caracter
cin.ignore(2); //ignora 2 caracteres
```

istream& read(char *s,int n);

Usada para ler n caracteres sem interpretar o conteúdo destes caracteres e armazenar em s.

Sentenças para istream

- O operador >>, ao ler uma stream (do teclado ou do disco) desconsidera espaços em branco ' ', nova linha '\n', avanço de formulário '\f' e retorno de carro (enter).
- A função isspace definida em <cctype> informa se é um espaço.

⁷Observe que o operador >> não é sobrecarregado para ler um único caracter. Para ler um único caracter use cin.get(ch).

- A leitura para um string do C [char*], automaticamente inclui o caracter de terminação ('\0').
- ²Sobrecarga de uma istream

```
Exemplo
istream& operator>> (istream& in, Tipo obj)
{
in >> obj.atributo;
return in;
}
```

Portabilidade: O caracter usado para encerrar uma entrada de dados é diferente nas plataformas DOS/Windows e Unix/Linux/Mac.

```
Exemplo:
//No DOS um ctrl+z encerra a função abaixo
//No Unix um ctrl+d encerra a função abaixo
int var;
do
{
cout << "Entre com um número:";
}while( cin >> var;);
```

Dica³: Existe uma relação entre cin e cout, observe no exemplo acima que cout não esta enviando um caracter de nova linha ('\n'), mas ao executar o programa a nova linha é incluída ?. É que o objeto cin envia para cout uma nova linha.

```
Exemplo:
//Lê os caracteres do teclado e
//joga para tela até que ctrl+d seja digitado.
char c;
while(c = cin.get())
    cout.put(c);
```

23.7 A classe <ostream>

A classe <ostream> é utilizada para saída de dados. C++ cria automaticamente o objeto cout que pode ser utilizado para saída na tela do computador. Observe que cout é uma abreviação de C out.

A formatação da saída para a tela é realizada em dois passos.

Primeiro define-se os atributos do objeto ostream (alinhamento, espaçamento, formatos). A seguir envia-se a stream para a saída desejada.

Métodos de <ostream>**ostream& flush();**Descarrega o bufer⁸, ou seja, enviar imediatamente os caracteres para o seu destino.**ostream& put(char ch);**

Insere o caracter ch na stream.

long tellp();

Retorna a posição do ponteiro put.

ostream & write(const signed char* s,streamsize n);

Envia para a saída a string s com até n caracteres. Não interpreta o conteúdo de s.

Veja na Tabela 23.3 os caracteres de escape. Quando inseridos em uma stream, realizam uma determinada tarefa especial.

Tabela 23.3: Caracteres de escape.

Caracter	Efeito
'\a'	Toca o alarme(beep)
'\b'	Retrocede uma coluna(retrocesso)
'\f'	Próxima linha, ou pagina(ff)
'\n'	Próxima linha, nova linha
'\r'	Retorno de carro
'\t'	Tabulação horizontal
\v	Tabulação vertical
'\o'	Caractere nulo, usado no fim da string
'\\'	Imprime \
'\''	Imprime '

As operações de saída para tela podem ser realizadas com o objeto cout⁹ e o operador de inserção. O exemplo esclarece.

Listing 23.3: Formatação da saída de dados usando iomanip.

```
//Arquivo ex-Entrada-Saida3.cpp
#include <iostream>
#include <string>
using namespace std;

int main()
{
    int    i    = 5;
    double d    = 1.23456789;
```

⁸Como dito anteriormente, uma stream tem um bufer onde os caracteres ficam armazenados. Quando o bufer enche, os caracteres são descarregados, isto é, enviados para o seu destino. O método flush solicita o descarregamento imediato do bufer.

⁹Observe que cout é um acrônimo para C output.

```

string nome = "ClubePalestraItália-Palmeiras";
char letra = 'c';
char cstring[] = "STRING_DE_C";

cout << "-----Formato padrão-----" << endl;
cout << "int_i double_d string_nome char_c char*_cstring" << endl;
cout << "-----" << endl;
cout << i << " " << d << " " << nome << letra << " " << cstring << endl;

cout << "Alinhamento a esquerda" << endl;
cout.setf(ios::left);
cout.width(10);
cout << i << " ";
cout.width(10);
cout << d << endl;

cout << "Alinhamento a direita" << endl;
cout.setf(ios::right);
cout.width(10);
cout << i << " " << d << endl;

cout << "Formato científico" << endl;
cout.setf(ios::scientific);
cout << i << " " << d << endl;

cout << "Mostra sinal positivo" << endl;
cout.setf(ios::showpos);
cout << i << " " << d << endl;

cout << "Maiúsculas" << endl;
cout.setf(ios::uppercase);
cout << nome << " " << cstring << endl;

cout << "Ativa hexadecimal" << endl;
cout.setf(ios::hex, ios::basefield);
cout << i << " " << d << " " << nome << endl;

cout << "Desativa hexadecimal" << endl;
cout.unsetf(ios::hex);
cout << "16=" << d << endl;

cout << "Seta precisão numérica em 12" << endl;
cout.precision(12);
cout << i << " " << d << " " << nome << endl;

cout << "Seta espaço do campo em 20 caracteres, temporário" << endl;
cout.width(20);
cout << i << " " << d << " " << nome << endl;

cout << "Caracter de preenchimento '#' " << endl;
cout.fill('#'); cout.width(20);
cout << i << " " << d << " " << nome << endl;

cout << "Escreve na tela 5 caracteres da cstring" << endl;
cout.write(cstring, 5);

```

```

cout << endl;

cout << "Imprime letrana tela" << endl;
cout.put('G');
cout<<endl;

cout << "Imprime a letra" << endl;
cout << letra << endl;

cout << "Imprime o endereço da letra"<< endl;
cout << & letra <<endl;

cout << "Imprime código ascii da letra"<< endl;
cout << (int)letra << endl ;

cout << "Imprime a variável int em hexadecimal" << endl;
cout << hex << i <<endl;

cout << "Parenteses evita ambiguidade, imprime \"9\"" <<endl;
cout << (5+4)<<endl;

cout << "Imprime string de C" << endl;
cout << cstring << endl;
/*
bool b = false;
cout <<"b="<<b<<endl;
cout << boolalpha <<"b="<<b<<endl;
*/
//Usado para descarregar o buffer
cout.flush();

return 0;
}

/*
Novidade:
-----
Mostra o uso da formatação da saída de dados em C++
*/

/*
Saída:
-----
[andre@mercurio Cap3-P00UsandoC++]$ ./a.out
-----Formato padrão-----
int_i double_d string_nome char_c char*_cstring
-----
5 1.23457 Clube Palestra Itália - Palmeiras c STRING_DE_C
Alinhamento a esquerda
5 1.23457
Alinhamento a direita
5 1.23457
Formato científico
5 1.234568e+00
Mostra sinal positivo

```

```

+5 +1.234568e+00
Maiúsculas
Clube Palestra Itália - Palmeiras  STRING_DE_C
Ativa hexadecimal
5 +1.234568E+00  Clube Palestra Itália - Palmeiras
Desativa hexadecimal
16= +16
Seta precisão numérica em 12
+5 +1.234567890000E+00  Clube Palestra Itália - Palmeiras
Seta espaço do campo em 20 caracteres, temporário
+5 +1.234567890000E+00  Clube Palestra Itália - Palmeiras
Caracter de preenchimento '*'
#####+5 +1.234567890000E+00  Clube Palestra Itália - Palmeiras
Escreve na tela 5 caracteres da cstring
STRIN
Imprime letra G na tela
G
Imprime a letra
c
Imprime o endereço da letra
c0
Imprime código ascii da letra
+99
Imprime a variável int em hexadecimal
5
Parenteses evita ambiguidade, imprime "9"
9
Imprime string de C
STRING_DE_C

*/

```

Sentenças para ostream

- O valor na saída de cout é arredondado e não truncado.
- eof() retorna true (1) se estamos no final do arquivo e false (0) caso contrário.
- Um bool tem como saída 0 ou 1. Para ter na saída “false” e “true”, basta ativar este tipo de saída usando:

```
cout << boolalpha;
```

- Sobrecarga de uma ostream

```

Exemplo
ostream& operator<< (ostream& out, Tipo obj)
{
out << obj.atributo;
return out;
}

```

- ²Observe que a sobrecarga de >> e << é realizada com funções friend, não admitindo o uso do polimorfismo (não são métodos virtuais da classe). Mas você pode criar um método virtual para entrada e saída de dados (ex: virtual void Entrada(); virtual void Saida()).
- ²Como os operadores << e >> são muito usados, pense em declará-los como inline.

23.8 A classe <sstream>

O arquivo <sstream> inclui as classes ostream e istream. As mesmas representam um misto de uma classe string e uma classe stream, funcionando, ora como uma string, ora como uma stream.

As classes ostream e istream¹⁰ podem ser utilizadas para substituir com vantagens a função printf de C.

O exemplo abaixo mostra como formatar uma string usando as funções de ostream e istream.

Listing 23.4: Uso de sstream (ostream e istream).

```
# include <iostream>    //streams
# include <string>      //string
# include <sstream>     //stream e string junto
# include <fstream>
using namespace std;

main()
{
//Cria objetos
  string s1("oi_tudo"),s2("bem");
  double d=1.2345;
  int i=5;

  //Cria ostream com nome os
  //funciona ora como stream ora como string
  ostream os;

  //abaixo "os" funciona como uma stream, como cout
  os << s1 <<" " << s2 <<" " << d <<" " << i;

  //abaixo os funciona como uma string
  cout << "os.str()=" << os.str() << endl;

  //Cria objeto do tipo istream com nome in
  //Aqui in funciona como uma string
  istream in( os.str() );

  //Cria strings s3 e s4 e dados numéricos d2 e i2
  string s3, s4; double d2; int i2;

  //na linha abaixo, joga da stream in para os dados numéricos,
  //aqui in funciona como uma stream
  in >> s3 >> s4 >> d2 >> i2;
```

¹⁰ostream = output string stream, e istream = input string stream.

```

cout<< "s3_=" << s3 << "\n"
      << "s4_=" << s4 << "\n"
      << "d2_=" << d2 << "\n"
      << "i2_=" << i2 << endl;
}

/*
Novidade:
Uso de objeto ostream para formatar um nome de arquivo de disco
Uso de stringstream para converter string em numeros.
Observe que s1 e s2 não tem espaços em branco
*/
/*
Saída:
-----
[andre@mercurio Cap3-P00UsandoC++]$ ./a.out
os.str()=oi_tudo bem 1.2345 5
s3 = oi_tudo
s4 = bem
d2 = 1.2345
i2 = 5
*/

```

Preste atenção no exemplo acima. Primeiro criamos um objeto `os` que é usado para armazenar os caracteres das strings `s1`, `s2` e os números `d` e `i`, funcionando como uma stream (como `cout`). A seguir, o objeto `os` é usado para mostrar a string na tela com `os.str()`, funcionando como uma string.

Você pode usar um objeto `os` para formatar uma saída de dados em uma string. Observe que a função `setw` e os manipuladores de `<iomanip>` podem ser utilizados.

No final do exemplo cria-se um objeto `in`, que recebe como entrada a string formatada em `os.str()`, ou seja, o objeto `in` atua como uma string. A seguir `in` é usado para enviar os caracteres para as strings `s3`, `s4` e os números `d2` e `i2`, atuando como uma stream (como `cin`).

Observe que com um objeto do tipo `ostream`, pode-se substituir com vantagens a antiga função `sprintf` de C e com um objeto `istream` substituir a antiga `fscanf`.

23.9 Sentenças para stream

- Nas suas classes inclua sobrecarga para os operadores `>>` e `<<`.
- Para ter certeza da saída do `<<`, use parenteses.

Exemplo:

```
cout << (a+b);
```

- Após uma leitura com `cin`, use `cin.get()` para retirar o caracter de return.
- Lembre-se que `width` se aplica a próxima saída.

Capítulo 24

Entrada e Saída com Arquivos de Disco

Neste capítulo apresenta-se as classes `<fstream>`, `<ifstream>` e `<ofstream>`, a mesmas fornecem acesso a arquivos de disco.

24.1 Introdução ao acesso a disco

Herdeira da classe `ostream`, a `ofstream` é usada para enviar caracteres para um arquivo de disco. Herdeira da classe `istream`, a `ifstream` é usada para ler caracteres de um arquivo de disco. Herdeira da `iostream`, a `fstream` é usada para leitura e escrita em arquivos de disco. Reveja a Figura 23.1, que ilustra a hierarquia destas classes.

24.2 A classe `<fstream>`

Métodos de `<fstream>`

`fstream()`;

Construtor da classe, cria objeto sem associá-lo a um arquivo de disco.

`fstream(const char* arq, int=modo abertura, int=modo de proteção);`

Construtor sobrecarregado, cria objeto e associa a arquivo com nome `arq`.

modo de abertura, especifica como abrir o arquivo (veja Tabela 24.1)

modo de proteção, especifica o formato de proteção do arquivo (veja Tabela 24.2)

`fstream(int);`

Construtor sobrecarregado, cria objeto e associa a arquivo identificado com um número `int`.

`void close(void);`

Descarrega o bufer e fecha o arquivo aberto, se já fechado ignora.

`void open (const char *arquivo, int modo_de_abertura, int modo_de_proteção);`

O método `open` é usado para abrir arquivos de disco.

nome_do_arquivo é o nome do arquivo,

modo de abertura, especifica como abrir o arquivo (veja Tabela 24.1)

modo de proteção, especifica o formato de proteção do arquivo (veja Tabela 24.2).

Tabela 24.1: Modos de abertura do método open.

ios::app	Acrescenta ao fim do arquivo
ios::ate	Vai para o fim do arquivo
ios::in	Abre o arquivo para entrada (leitura)
ios::out	Abre arquivo para saída (escrita)
ios::nocreat	Não cria se o arquivo não existe (uma falha)
ios::noreplace	Se o arquivo já existir ocorre uma falha
ios::binary	Abre o arquivo em modo binário
ios::trunc	Elimina o arquivo se já existe e recria

Tabela 24.2: Modos de proteção do método open (atributos de arquivo).

0	arquivo
1	apenas leitura
2	escondido
4	sistema
8	ativar o bit do arquivo (backup)

Listing 24.1: Uso de stream de disco (ifstream e ofstream) para escrever e ler em arquivos de disco.

```
//Arquivo ex-Entrada-Saida-fstream.cpp

#include <fstream>
#include <string>
#include <sstream>

using namespace std;

int main()
{
    {
        //Podemos criar o objeto e depois ligá-lo a um arquivo de disco
        //cria objeto do tipo fstream com nome fout
        ofstream fout;

        //Associa o objeto fout ao arquivo data.dat
        //um ofstream é um fstream com ios::out
        fout.open("data.dat");
        fout<<"Isto vai para o arquivo data.dat\n";

        //descarrega o bufer e fecha o arquivo.
        fout.close();
    }
    {
        //Podemos criar o objeto e já ligá-lo a um arquivo de disco
        ofstream fout("data2.dat");
        fout<<"Isto vai para o arquivo data2.dat\n";

        //descarrega o bufer e fecha o arquivo.
    }
}
```



```

    fout.close();
}

{
    //Um ifstream é um fstream com ios::in
    //associa objeto fin ao arquivo data.dat
    ifstream fin("data.dat");

    //Lê a string s do arquivo de disco data.dat
    string s;
    getline(fin, s);
    cout << "Lido o arquivo:" << s << endl;
    fin.close();
}

{
    //Cria objeto ostream com nome os
    //O tipo ostream é definido em sstream
    for ( int i = 0; i < 5 ; i++ )
    {
        ostream os;
        os << "nomeDoArquivo-" << i << ".dat";
        ofstream fout(os.str().c_str());
        fout << "no arquivo de disco:" << os.str() << endl;
        cout << "no arquivo de disco:" << os.str() << endl;
        fout.close();
    }
}

return 0;
}

/*
Novidade:
-----
Uso de arquivos de disco.
Como abrir um arquivo de disco.
Como escrever em um arquivo de disco.
Como ler de um arquivo de disco.
Como usar ostream para formatação do nome de arquivos sequenciais.
*/

/*
Saída:
-----
[andre@mercurio Cap3-P00UsandoC++]$ ./a.out
Lido o arquivo: Isto vai para o arquivo data.dat
no arquivo de disco: nomeDoArquivo-0.dat
no arquivo de disco: nomeDoArquivo-1.dat
no arquivo de disco: nomeDoArquivo-2.dat
no arquivo de disco: nomeDoArquivo-3.dat
no arquivo de disco: nomeDoArquivo-4.dat
*/

/*
Apresenta-se a seguir Os arquivos gerados:
PS: o comando cat mostra o conteúdo de um arquivo.
*/

```

```
[andre@mercurio Cap3-P00UsandoC++]$ cat data.dat
Isto vai para o arquivo data.dat

[andre@mercurio Cap3-P00UsandoC++]$ cat data2.dat
Isto vai para o arquivo data2.dat

[andre@mercurio Cap3-P00UsandoC++]$ cat nomeDoArquivo-0.dat
no arquivo de disco: nomeDoArquivo-0.dat

[andre@mercurio Cap3-P00UsandoC++]$ cat nomeDoArquivo-1.dat
no arquivo de disco: nomeDoArquivo-1.dat

[andre@mercurio Cap3-P00UsandoC++]$ cat nomeDoArquivo-2.dat
no arquivo de disco: nomeDoArquivo-2.dat

[andre@mercurio Cap3-P00UsandoC++]$ cat nomeDoArquivo-3.dat
no arquivo de disco: nomeDoArquivo-3.dat

[andre@mercurio Cap3-P00UsandoC++]$ cat nomeDoArquivo-4.dat
no arquivo de disco: nomeDoArquivo-4.dat
*/
```

24.3 Armazenando e lendo objetos

Com os métodos `read` e `write` podemos armazenar um objeto em disco e no outro dia ler este objeto do disco, o que é muito útil para manipulação de dados.

Protótipo:

```
istream & read(char *charbuffer, streamsize numerobytes);
ostream & write(const signed char *charbuffer, int numerobytes);
```

A listagem a seguir mostra um exemplo.

Listing 24.2: Leitura e gravação de objetos simples usando `read/write`.

```
#include <string>
#include <fstream>
#include <vector>

using namespace std;

//Declara classe Data
class Data
{
    int x;
    int y;

public :

    //construtor
    Data():x(0),y(0){};

    //sobrecarga operadores << e >> como funções friend.
```

```
//observe que Data é declarado como referencia
friend istream& operator >> (istream&, Data&);
friend ostream& operator << (ostream&, Data&);

//observe que Data é declarado como ponteiro
friend ifstream& operator >> (ifstream&, Data*&);
friend ofstream& operator << (ofstream&, Data*&);
};
//int Data::nobj=0;
int main()
{
//Solicita nome do arquivo de disco
cout << "Nome do Arquivo: ";
string nome_arquivo;
getline(cin,nome_arquivo);

//Abre arquivo de disco para escrita
ofstream fout (nome_arquivo.c_str());
if (! fout)
{
cout << "\n\nErro na Abertura de arquivo";
exit(1);
}

//Cria vetor para objetos do tipo Data
vector< Data > d;

//Cria objeto e ponteiro para objeto
Data obj;
Data* pObj;
pObj = &obj;

//Lê objeto e armazena em obj
cout <<"Entre com os valores de x e y de cada objeto. Para encerrar ctrl+d"
<<endl;
while(cin >> obj)
{
//armazena dados do objeto no arquivo de disco
fout << pObj;

//mostra objeto lido na tela
cout << "Objeto="<<obj;

//armazena no vetor
d.push_back(obj);
};

//reseta o stream cin
cin.clear();

//fecha o arquivo de disco
fout.close();

//mostra todos os objetos lidos
```

```

cout<<"\nMostrando objetos do vetor d\n" << endl;
for (int i = 0 ; i < d.size() ; i++)
    cout << d[i] << endl;

//abre arquivo de disco para leitura
cout << "Vai ler os objetos do disco" << endl;
ifstream fin (nome_arquivo.c_str());

//testa se ok
if( ! fin )
{
    cout << "\n\nErro na Abertura de arquivo";
    exit(1);
}

//cria um segundo vetor
vector< Data > d2;

//enquanto estiver lendo do arquivo (nã chegou ao final do arquivo)
//lê dados do objeto e armazena em obj
while(fin >> pobj)
{
    cout << obj;
    d2.push_back(obj);
    //obj.nobj++;
};
fin.close();

//mostra todos os objetos lidos
cout<<"\nMostrando objetos do vetor d2\n" << endl;
for (int i=0;i< d2.size();i++)
    cout << d2[i]<<endl;

cin.get();
return 0;
}

istream& operator >> (istream& in, Data& d)
{
    cout << "\nx: "; in >> d.x;
    cout << "\ny: "; in >> d.y;
    in.get();//retorno de carro
    return in;
}

ostream& operator << (ostream& out, Data& d)
{
    out << "(x= " << d.x ;
    out << ",y= " << d.y << ")" << endl;
    return out;
}

ifstream& operator >> (ifstream& in, Data*& d)
{

```

```

    in.read ((char*) d, sizeof(Data));
    return in;
}

ofstream& operator << (ofstream& out, Data*& d)
{
    //out.write((char *) d, d->nobj*sizeof(Data));
    out.write((char *) d, sizeof(Data));
    return out;
}

/*
Saída:
-----
[andre@mercurio Cap3-P00UsandoC++]$ g++ LeituraGravacaoDeObjetosSimples.cpp
[andre@mercurio Cap3-P00UsandoC++]$ ./a.out
Nome do Arquivo : Teste.txt
Entre com os valores de x e y de cada objeto. Para encerrar ctrl+d

x : 1
y : 2
Objeto=(x = 1, y = 2)

x : 3
y : 4
Objeto=(x = 3, y = 4)

x : 5
y : 6
Objeto=(x = 5, y = 6)

x :
y :
Mostrando objetos do vetor d
(x = 1, y = 2)
(x = 3, y = 4)
(x = 5, y = 6)

Vai ler os objetos do disco
(x = 1, y = 2)
(x = 3, y = 4)
(x = 5, y = 6)

Mostrando objetos do vetor d2
(x = 1, y = 2)
(x = 3, y = 4)
(x = 5, y = 6)
*/

```

Observe que este mecanismo de armazenamento e leitura de arquivos em disco funciona corretamente. Mas se o objeto usar alocação dinâmica para atributos internos, o que vai ser armazenado em disco é o conteúdo do ponteiro. Você terá de usar um outro mecanismo para gerenciar o armazenamento e leitura de objetos dinâmicos.

24.4 Posicionando ponteiros de arquivos com seekg(), seekp(), tellg(), tellp()²

Quando estamos lendo informações de um arquivo em disco, estamos com o nosso objeto de leitura apontando para uma determinada posição do arquivo de disco. Existem métodos que são usados para mudar a posição do ponteiro de leitura e do ponteiro de escrita.

Os métodos `seekg`, `seekp`, são utilizadas para posicionar o ponteiro de leitura (`get`) e de escrita (`put`) em um determinado arquivo. As funções `tellg` e `tellp` são utilizadas para obter a posição dos ponteiros `get` e `put`, respectivamente. Veja a seguir o protótipo dos métodos `seekg` e `seekp` e na Tabela 24.3 os manipuladores que podem ser passados para estes métodos.

Protótipos:

```
istream & seekg(streamoff offset, seekdir org);  
ostream & seekp(streamoff offset, seekdir org);
```

Exemplo:

```
//movimenta ponteiro get  
fin.seekg(deslocamento,manipulador);  
//movimenta ponteiro put  
fout.seekp(deslocamento,manipulador);  
//obtem posição do ponteiro get  
fin.tellg();  
//obtem posição do ponteiro put  
fout.tellp();
```

Tabela 24.3: Manipuladores para os métodos `seekp` e `seekg`.

<code>basic_ios::beg</code>	vai para o início do arquivo
<code>basic_ios::end</code>	vai para o fim do arquivo
<code>basic_ios::cur</code>	posição corrente
<code>basic_ios::fail</code>	operação i/o falhou, arquivo estragado
<code>basic_ios::bad</code>	i/o inválida, ou arquivo estragado

Exemplo:

```
//Descreve a classe A  
class A  
{public:  
A():x(0),y(0){};  
int x; int y;  
void Input()  
{  
cout << "Entre com x e y (x espaço y):";  
cin >> x >> y; cin.get();  
}
```

```

    }
};
//Armazena objetos em disco
void Armazena_objetos()
{
//cria 5 objetos estáticos
vector < A > obja(5);
//cria o objeto fout, que aponta para o arquivo readwrite.dat
ofstream fout("readwrite.dat");
for(int i = 0; i < obja.size() ; i++)
    {
//entrada de dados do objeto A[i]
A[i].Input();
//armazena objeto i, usando função write
fout.write((char*)obja[i],sizeof(obja[i]));
    }
}
void Le_objetos()
{
//cria 5 objetos estáticos
vector< A > objb(5);
//cria objeto fin, que aponta para arquivo readwrite.dat
ifstream fin("readwrite.dat");
cout << "Qual objeto quer ler?";
int i;
cin >> i; cin.get();
//vai até a posição inicial do objeto i no disco
fin.seekg(i*sizeof(objb[i])+1,ios::beg)
//lê o objeto i, usando método read
fin.read((char*)objb[i],sizeof(objb[i]));
//mostra atributos do objeto lido
cout <<"objb["<i<<"].x= " << objb[i].x
    <<"objb["<i<<"].y= " << objb[i].y << endl;
}

```

24.5 Acessando a impressora e a saída auxiliar³

Para ligar um arquivo diretamente a impressora utilize:

- `fstream cprn(4);` //cprn é conectado a impressora
- `fstream caux(3);` //caux é conectado a saída auxiliar

Exemplo:

```

fstream cprn(4);
cprn <<"Estou escrevendo na impressora";

```

```
cprn.close();
```

Exemplo:

```
fstream caux(3);
caux << "Estou enviando texto para saída auxiliar";
caux.close();
```

24.6 Arquivos de disco binários³

Funções que operam byte a byte, sem interpretar estes bytes.

Protótipos:

```
istream & get(char & c);
ostream & put(char & c);
```

Exemplo:

```
string s = "oi tudo bem";
//enquanto houver algo
while(s.at[i])
    //escreve na tela
    cout.put(s[i++]);
cout << end;
//enquanto houver algo escreve na tela
while(cin.get(caracter))
    cout << caracter;
```

24.7 Executando e enviando comandos para um outro programa

Um outro exemplo muito interessante e útil do uso das streams de C++ é a execução e o envio de comandos para um outro programa. Isto é, o seu programa pode executar um programaB e enviar comandos para o programaB.

No exemplo apresentado na listagem a seguir, vai executar o programa `gnuplot`¹.

Listing 24.3: Executando e enviando comandos para um outro programa (com `ofstream`).

```
#include <cstdio>
#include <cmath>
#include <fstream>
#include <pfstream.h>
using namespace std;

void main()
{
    ofstream fout("data.dat");
```

¹Um programa usado para plotar gráficos, muito simples e útil. Disponível na maioria das distribuições Linux. Para outras plataformas, isto é, Windows, Mac OS X, consulte o site da gnu.


```

float x,y,z;
for ( x =-5; x <= 5 ; x += 0.1 )
{
    y = x * x * x - 1.5 * x * x + 7;
    z = x * sin(x);
    fout << x << " " << y << " " << z <<endl;
}
fout.close();

ofstream gnuplot ("|gnuplot");
gnuplot << "plot 'data.dat' using 1:2 title 'dados de y' with linespoint"
    << " , 'data.dat' using 1:3 title 'dados de z' with linespoint" <<endl
;
gnuplot.flush();
cout << "\nPressione enter" <<endl;
cin.get();
gnuplot.close();
}

/*
Novidade:
=====
Uso de ofstream para executar um programa externo e enviar comandos
diretamente para este programa.

Neste exemplo, vai executar o programa gnuplot
e enviar para o programa gnuplot o comando
"plot 'data.dat' with linespoint\n"

Dica: o gnuplot é executado porque ofstream trata a barra |
como uma instrução para executar o programa |gnuplot e não abrir um
arquivo de disco.
Ou seja, você precisa incluir antes do nome do programa a ser executado
uma barra (PS; testado na plataforma Linux).
*/

```

24.8 Redirecionamento de entrada e saída

Tanto na linha de comando do Linux, como do Mac OS X, como do DOS, você pode usar os mecanismos de redirecionamento de entrada e saída de dados.

Você já usou o comando `ls` (ou `dir`) para listar o conteúdo do diretório.

Exemplo:
`ls`

Entretanto, se a saída do comando `ls` for muito grande e não couber na tela, os primeiros diretórios e arquivos irão desaparecer. Nestes casos, você pode usar um comando de redirecionamento como pipe (`|`), para enviar a saída do programa `ls` para um paginador como o `less`². Veja o exemplo.

²O programa `less` é um paginador, permite que você navegue pela listagem de diretório. Para sair do `less` digite a letra `q`.

Exemplo:

```
ls | less
```

Ou seja, você pega a saída do programa ls e envia para o programa less.

Quando você executa um programa, a saída padrão é a tela. Você pode redirecionar a saída do programa para um arquivo de disco usando >, veja o exemplo.

Exemplo:

```
ls > arquivo.dat
cat arquivo.dat
```

No exemplo acima pega a saída do programa ls e envia para o arquivo arquivo.dat. O programa cat apenas mostra o conteúdo do arquivo.dat.

Se o arquivo arquivo.dat for muito grande, use novamente o paginador less.

Exemplo:

```
cat arquivo.dat | less
```

Você pode adicionar a saída de um programa no final de um arquivo usando o operador de concatenação. Ou seja, escrever a saída de um programa em um arquivo de disco já existente.

Exemplo:

```
./Simulacao > resultado.dat
./Simulacao >> resultado.dat
```

Neste caso primeiro cria o arquivo resultado.dat que armazenará o resultado da primeira simulação realizada. Depois executa novamente o programa de Simulacao, adicionando os novos resultados no final do arquivo resultado.dat.

Outra possibilidade bastante interessante é criar um arquivo de disco com as entradas que serão solicitadas por determinado programa. Assim, em vez de esperar o usuário digitar cada entrada, o programa lê as entradas diretamente do arquivo de disco.

Exemplo:

```
./Simulacao < entrada.dat
```

Listing 24.4: Usando redirecionamento de arquivo.

```
#include <fstream>
#include <string>

void main()
{
    //Exemplo que lê os dados de uma simulação diretamente
    //de arquivo de disco usando redirecionamento de entrada.
    string nomeArquivoDisco;
    cout << "Entre com o nome do arquivo de disco: ";
    getline(cin, nomeArquivoDisco);

    cout << "Entre com o número de repetições: ";
```

```

int repeticoes=0;
cin >> repeticoes; cin.get();

cout << "Entre com a precisão do solver: ";
double precisao = 0.0001;
cin >> precisao; cin.get();

cout << "VALORES ENTRADOS / LIDOS" << endl;
cout << "nome do arquivo de disco = " << nomeArquivoDisco << endl ;
cout << "número de repetições = " << repeticoes << endl ;
cout << "precisao = " << precisao << endl ;
}

```

/*

Novidade:

Uso de redirecionamento de entrada.

Neste programa você pode entrar os dados via teclado ou usando um arquivo de disco e o redirecionamento:

Você precisa:

De um arquivo de disco com o nome: dados_simulacao.dat e o conteúdo:

NomeArquivoDisco.dat

1

0.0004

**/*

/*

Saída - Entrando os dados via teclado:

[andre@mercurio Cap3-P00UsandoC++]\$./a.out

Entre com o nome do arquivo de disco: teste.dat

Entre com o número de repetições: 3

Entre com a precisão do solver: .123

VALORES ENTRADOS / LIDOS

nome do arquivo de disco = teste.dat

número de repetições = 3

precisao = 0.123

Saída - Entrando os dados via redirecionamento:

[andre@mercurio Cap3-P00UsandoC++]\$./a.out < dados_simulacao.dat

Entre com o nome do arquivo de disco: Entre com o número de repetições:

Entre com a precisão do solver:

VALORES ENTRADOS / LIDOS

nome do arquivo de disco = NomeArquivoDisco.dat

número de repetições = 1

precisao = 0.0004

**/*

Capítulo 25

class <string>

Apresenta-se neste capítulo a classe `string` de C++. A mesma já foi utilizada em nossos exemplos, sem maiores explicações. Neste capítulo vamos nos aprofundar no uso da classe `string`. Discute-se os diferentes construtores de `string`, e os métodos utilizados para modificar uma `string` de C++.

Basicamente uma `string` é uma sequência de caracteres.

A classe `string` é atualmente uma classe `template` que pode manipular caracteres de 8-bits¹ ASCII bem como caracteres de 16-bits, ou seja, foram definidos os `typedef`'s:

- `typedef basic_string<char, string_traits<char> > string;`
- `typedef basic_string<wchar_t> wstring;`

Para utilizar a classe `string`, inclua o arquivo de cabeçalho `<string>`.

Exemplo:

```
# include <string>
```

Descreve-se a seguir os diferentes construtores fornecidos pela classe `string`.

Construtores

Exemplo:

```
//Cria string com nome s1
string s1;
//Cria string com nome s2 e armazena "a classe string"
string s2 ("a classe string");
//Cria string com nome s3 e inicializa com
string s3 = " é legal";
//Cria string com nome s4 uma cópia de s3 (usa construtor de cópia)
string s4 (s3);
//Cria string com nome s6 e define tamanho como sendo 100 caracteres
string s6 ("eu tenho espaço para 100", 100);
//Cria string s7, com espaço para 10 letras, preenche com b
string s7 (10, 'b');
```

¹Lembre-se, 1 byte = 8 bit.

```
//Cria string s8, uma cópia de s6 (usa iteradores)2
string s8 (s6.begin(), s6.end());
```

Manipulação do tamanho da string:

A classe string tem um conjunto de métodos para manipulação do tamanho da string. Pode-se obter o tamanho usado (size), e a capacidade real da string (capacity). A função max_size retorna o tamanho da maior string que pode ser construída. Pode-se redimensionar a string com resize.

Atribuição e acesso:

Inserção, remoção e substituição:

Você pode remover ou substituir pedaços da string. Pode inserir novos caracteres ou sub-strings em uma string existente.

Substrings:

Pode-se criar e manipular substrings a partir de uma string.

Find e rfind:

A classe string fornece funções de pesquisa com find e pesquisa invertida com rfind.

A função find() determina a primeira ocorrência na string, pode-se especificar com um inteiro a posição inicial da busca. A função rfind() busca da posição final para a inicial, ou seja, pesquisa reversa.

Outras funções de pesquisa find_first_of(), find_last_of(), find_first_not_of(), e find_last_not_of(), tratam a string argumento como um conjunto de caracteres, a posição do primeiro caracter encontrado é retornada, se não encontrar retorna um out_of_range.

Veja na listagem a seguir, o uso da classes string. Os diversos métodos acima são testados.

Listing 25.1: Uso de string.

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

void main ()
{
//Cria string com nome s1
string s1;

//Cria string com nome s2 e armazena "a classe string"
string s2 ("a classe string");

//Cria string com nome s3 e inicializa com
string s3 = "é legal";

//Cria string com nome s4 uma cópia de s3 (usa construtor de cópia)
string s4 (s3);
```

²Os iteradores serão discutidos no capítulo 28.3.

```

    string s5 (s3);

    //Cria string com nome s6 e define tamanho como sendo 100 caracteres
    string s6 ("eu_tenho_espaço_para_100", 100);

    //Cria string s7, com espaço para 10 letras, preenche com b
    string s7 (10, 'b');

    cout << s1 << "\n" << s2 << "\n" << s3 << "\n_" << s4 << "\n" << s5 << "\n"
        << s6 << "\n" << s7 << endl;

    //tamanho corrente da string e capacidade corrente da string
    cout << "_s6.size()" << s6.size () << endl;
    cout << "_s6.capacity()" << s6.capacity () << endl;

    //altera a capacidade da string
    s6.reserve (200);
    cout << "_Após_s6.reserve(200);_>_s6.size()" << s6.size () << endl;
    cout << "_s6.capacity()" << s6.capacity () << endl;

    //tamanho máximo da string que pode ser alocada
    cout << "s6.max_size()" << s6.max_size () << endl;

    //redimensiona a string e preenche com o caracter 't'
    cout << "s7.size()" << s7.size () << "_s7=_" << s7 << endl;
    s7.resize (15, 't');
    cout << "depois_s7.resize(15,_'t');_s7.size()" << s7.size () << endl;

    //tamanho corrente da string (15), o mesmo que size()
    cout << "s7.length()" << s7.length () << endl;

    //retorna true se estiver vazia
    if (s7.empty ())
        cout << "string_s7_vazia" << endl;

    //cópia de strings
    s1 = s2;
    cout << "s1=" << s1 << "\ns2=" << s2 << endl;

    //atribuição de uma string padrão de C
    s2 = "um_dois_três";

    //atribuição de um único caracter
    s3 = 'q';
    cout << "\ns2=" << s2 << "\ns3=" << s3 << endl;

    //adicionar a string existente (concatenar)
    s3 += "uatro";

    //define os três primeiros caracteres de s4 a partir de s2
    s4.assign (s2, 3);
    cout << "s3=" << s3 << "\ns4=" << s4 << endl;

    //define os caracteres 2, 3 e 4

```

```

s4.append (s5, 2, 3);
cout << "após s4.append(s5, 2, 3); s4=" << s4 << endl;

//cria uma cópia de s2, adiciona s3, e mostra na tela
cout << "(s2+s3)=" << (s2 + s3) << endl;

//troca o conteúdo das strings s4 e s5
cout << "s4=" << s4 << "\ns5=" << s5 << endl;
s5.swap (s4);
cout << "s4=" << s4 << "\ns5=" << s5 << endl;

//acessa a posição 2 da string (como em vetores)
cout << "s4.lenght()=" << s4.size () << endl;
cout << "s4[2]=" << s4[2] << endl;

//coloca a letra x na posição 2
s4[2] = 'x';
cout << "s4[2]=" << s4[2] << endl;

//mesmo que s4[2], acessa posição 2 (verifica acesso)
cout << s4.at (2) << endl;

//O método c_str() cria uma string no padrão C
char d[256];

//copia a string s4 para d
strcpy (d, s4.c_str ());

//remove as posições 4 e 5
//s3.remove(4, 2);

//substitue as posições 4 e 5 por pqr
s3.replace (4, 2, "pqr");
s2.insert (s2.begin () + 2, s3.begin (), s3.end ());
//s2.remove(s2.begin()+3, s2.begin()+5);
s2.replace (s2.begin () + 3, s2.begin () + 6, s3.begin (), s3.end ());

//adiciona abc após a posição 3
s3.insert (3, "abc");

//coloca em s4 posições 2 até o fim de s3 (testar)
//s3.copy (s4, 2);

//coloca em s4 posições 2 a 3 de s5
//s5.copy (s4, 2, 3);

//cria uma substring de s4, da posições 3 até o final
cout << s4.substr (3) << endl;

//cria uma substring de s4, a partir da posição 3, 3 e 4
cout << s4.substr (3, 2) << endl;

//s1[0]=m, s1[1]=i,..
s1 = "mississippi";

```



```

    cout << s1.find ("ss") << endl;          // retorna 2
    cout << s1.find ("ss", 3) << endl;      // retorna 5
    cout << s1.rfind ("ss") << endl;       // retorna 5
    cout << s1.rfind ("ss", 4) << endl;    // retorna 2

//procura a primeira ocorr\~{e}ncia de aeiou
    int i = s2.find_first_of ("aeiou");

//próxima não vogal
    int j = s2.find_first_not_of ("aeiou", i);
    cout << "i=" << i << "uj=" << j;
}

/*
Novidade:
-----
Uso de objetos e métodos de string
*/
/*
Saída:
-----
[andre@mercurio Cap4-STL]$ ./a.out

a classe string
é legal
é legal
é legal
eu tenho espaço para 100

    s6.size()= s6.capacity()= Após s6.reserve(200); -> s6.
bbbbbbbbbbbb
    s6.size()=100
    s6.capacity()=128
    Após s6.reserve(200); -> s6.size()=100
    s6.capacity()=128
s6.max_size()=4294967294
s7.size()=10 s7= bbbbbbbbbbbb
depois s7.resize(15, 't'); s7.size()=15
s7.length()=15
s1=a classe string
s2=a classe string

s2=um dois três
s3=q
s3=quatro
s4=dois três
após s4.append (s5, 2, 3); s4=dois três le
(s2 + s3)=um dois trêsquatro
s4=dois três le
s5= é legal
s4= é legal
s5=dois três le
s4.lenght()=8
s4[2]=
s4[2]=x

```

```

x
legal
le
2
5
5
2
*/

```

Veja na listagem a seguir, o uso das classes string, sstream e ifstream para executar um programa do shell.

Listing 25.2: Uso de string e sstream para executar um programa do shell.

```

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

void main()
{
    //Lista dos arquivos com extensão jpg
    //pode substituir por comando como find...
    system("ls *.jpg > lixo");

    //Abre arquivo de disco
    ifstream fin("lixo");
    string arq;

    //Enquanto tiver algo no arquivo de disco, ler o nome do arquivo
    while ( fin >> arq )
    {
        //Determina posição do jpg
        int posicao = arq.find("jpg");
        //Cria os
        ostringstream os;
        //Inicio do comando "jpeg2ps arqin.jpg "
        os << "jpeg2ps_" << arq ;
        //Substitue extensão do arquivo por ps
        arq.replace(posicao,3,"ps");
        //Fim do comando " > arqout.ps"
        os << "_>" << arq;
        //Executa o comando
        cout << os.str() << endl << endl;
        system(os.str().c_str());
        //Elimina o arquivo lixo
        system("rm -f lixo");
    }
}

/*
Novidade:
-----
Interface de programa em C++ com programas do shell.
*/

```

25.1 Sentenças para strings

- `& s[0]` não é um ponteiro para o primeiro elemento da string.
- string não tem um ponteiro para o primeiro elemento.
- Se a string for redimensionalizada, possivelmente os iteradores existentes vão apontar para um monte de lixo.
- As funções `insert()` e `remove()` são similares as de vetores. A função `replace()` é uma combinação de `remove` e `insert`, substituindo o intervalo especificado por novo valor.
- A função `compare()` raramente é acessada diretamente, normalmente utiliza-se os operadores de comparação (`<`, `<=`, `==`, `!=`, `>=` and `>`). Pode-se comparar 2 strings ou uma string com uma string padrão `c`.

Dica: Releia o capítulo com atenção, releia cada linha e verifique como a mesma pode ser utilizada em seus programas.

Observe no exemplo a seguir o uso e as diferenças dos tipos `int` e `unsigned int`.

Capítulo 26

class <complex>

A classe `complex` é uma classe que suporta números complexos, com a parte real e imaginária.

Com a classe `complex` você pode trabalhar com números complexos diretamente, utilizando os operadores e métodos sobrecarregadas, isto é, pode-se igualar dois números complexos, fazer comparações, realizar operações aritméticas (+-*/), exponenciação, obter logaritmos, potência, entre outros.

Os objetos `complex` podem ter precisão `float`, `double` e `long double`.

Para criar um número complexo você precisa incluir o arquivo de cabeçalho `<complex>` e a seguir definir a precisão do número complexo.

```
Exemplo:  
#include <complex>  
complex <float> cf;  
complex <double> cd;  
complex <long double> cld;
```

Construtores

A classe `complex` representa um número complexo, ou seja, você pode construir um número complexo zerado, a partir da parte real, ou a partir da parte real e imaginária. Você também pode construir um número complexo a partir de outro. Veja a seguir um exemplo.

```
Exemplo:  
//Cria um número complexo, do tipo float com nome cf  
complex <float> cf;  
//Cria um número complexo e passa parte real  
double parteReal = 3.4;  
complex <double> cd( parteReal);  
//Cria um número complexo e passa a parte real e a parte imaginária  
double parteImg = 21.5;  
complex <long double> cld( parteReal,parteImg);  
//Construtor de cópia, cria cópia de cf  
complex <float> cf2(cf);
```

Métodos de acesso

A classe `complexo` fornece um conjunto de métodos que possibilitam a obtenção de propriedades do mesmo. Como o módulo, o argumento, a norma, o conjugado, entre outros.

Exemplo:

```
//retorna parte real
float real = cf.real();
//retorna parte imaginária
float img = cf.imag();
//Retorna módulo
float abs = cf.abs( );
//retorna argumento
float arg = cf.arg();
//soma dos quadrados da parte real e imaginária
float norm = cf.norm();
//a magnitude e o angulo de fase
const float magnitude =3; const float fase =45;
float p = cf.polar (magnitude, fase);
//retorna o conjugado
complex <float> cf_conjugado = cf.conj();
```

Métodos transcendentais

A classe `complex` fornece um conjunto de métodos transcendentais, como: `acos`, `asin`, `atan`, `atan2`, `cos`, `cosh`, `exp`, `log`, `log10`, `pow`, `sin`, `sinh`, `sqrt`, `tan`, `tanh`.

Operadores sobrecarregados

Diversos operadores foram sobrecarregados para a classe `complex`, a lista é dada por:

`+`, `-`, `*`, `/`, `+=`, `-=`, `*=`, `-=`, `*`, `+`, `-`

Operadores de comparação

`==`, `!=`,

Operadores de inserção e extração (stream)

```
template <class X> istream& operator >> (istream& is, complex<X>& x);
template <class X> ostream& operator << (ostream& os, const complex<X>& x);
```

Apresenta-se a seguir um exemplo de uso da classe `complex`.

Listing 26.1: Uso de `complex`.

```
//Exemplo: Uso da classe complex
#include <iostream>
# include <complex>

using namespace std;
int main ()
```

```

{

    complex < double >a (1.2, 3.4);
    complex < double >b (-9.8, -7.6);

    cout << "a= " << a << ", b= " << b << endl;

    a += b;

    b /= sin (b) * cos (a);

    cout << "a= " << a << ", b= " << b << endl;

    b *= log (a) + pow (b, a);

    a -= a / b;

    cout << "a= " << a << ", b= " << b << endl;
    cout << "Entre com o complexo a(real, imag): ";
    cin >> a;
    cin.get ();
    cout << "Conteúdo de a=" << a << endl;
    return 0;
}

/*
Novidade:
-----
Uso da classe complex
*/
/*
Saída:
-----
[andre@mercurio Cap4-STL]$ ./a.out
a = (1.2,3.4), b = (-9.8, -7.6)
a = (-8.6,-4.2), b = (7.77139e-05, -0.000364079)
a = (-8.6,-4.2), b = (3.37075e+23, -1.75436e+23)
Entre com o complexo a(real, imag): (23,123)
Conteúdo de a=(23,123)
*/

```


Capítulo 27

class <bitset>

A classe `bitset` é uma classe para manipulação de conjuntos de bits. O tamanho do vetor de bits deve ser definido em tempo de compilação, ou seja, um objeto `bitset` é um objeto estático.

Para criar um objeto `bitset` inclua o arquivo de cabeçalho `<bitset>`. O exemplo abaixo mostra como criar e usar um `bitset`, apresenta-se uma breve descrição de cada função de `bitset`.

Listing 27.1: Usando `bitset` - Exemplo 1

```
#include <bitset>
#include <iostream>
using std::cout;
using std::endl;

const int size = 5;

void Mostra (bitset < size > b);

int main ()
{
    //Cria objeto do tipo bitset com tamanho size e nome b.
    bitset < size > b;

    long unsigned int n = 2;

    //Seta o bit n para true
    b.set (n);
    b.set (n + 1);

    Mostra (b);

    //Seta o bit n para false
    b.reset (n);
    Mostra (b);

    //Seta todos os bits para false
    b.reset ();
    Mostra (b);

    //Inverte o bit n
    b.flip (n);
    Mostra (b);
}
```

```
//Inverte todos os bits
b.flip ();
Mostra (b);

//Retorna referência para o bit n, não verifica o intervalo do vetor
b[n];

//Retorna referência para o bit n, verifica o intervalo do vetor
//b.at(n);

//Retorna true se n esta no intervalo válido do vetor
bool
t =
b.
test (n);
cout << "bool_t_=_b.test(n);_t=_ " << t << endl;

//Tamanho do bitset
cout << "b.size()" << b.size () << endl;

//Número bits ativados
cout << "b.count()" << b.count () << endl;

//Retorna true se tem pelo menos 1 ativo
b.any ();
bool
f =
b.
none ();

//Retorna true se todos inativos
bitset < size > b1;
bitset < size > b2;
b1[1] = 1;

//Retorna true se o bitset b1 é todo igual a b2
if (b1 == b2)
cout << "b1==b2" << endl;

//Retorna true se o bitset b1 é diferente a b2
if (b1 != b2)
cout << "b1!=b2" << endl;

//Realiza um AND bit a bit e armazena em b1
b1[1] = 1;
b1 &= b2;
Mostra (b1);
Mostra (b2);

//Realiza um OR bit a bit e armazena em b1
b1[1] = 1;
b1 |= b2;
Mostra (b1);
```

```

    Mostra (b2);

//Realiza um XOR bit a bit e armazena em b1
    b1[1] = 1;
    b1 ^= b2;
    Mostra (b1);
    Mostra (b2);

//Rotaciona para direita n posições (todos os bits).
//Os bits iniciais assumem 0.
    b1[1] = 1;
    b1 >>= n;
    Mostra (b1);

//Rotaciona para esquerda n posições (todos os bits).
//Os bits finais assumem 0.
    b1 <<= n;
    Mostra (b1);

//Retorna uma string

    //cout<<"b.to_string()="<<b.to_string()<<endl;

//Retorna um ulong
    b.to_ulong ();
    return 1;
}

void
Mostra (bitset < size > b)
{
    for (int i = 0; i < b.size (); i++)
        cout << b[i] << "␣";
    cout << endl;
}

/*
Saída:
-----
[andre@mercurio Cap4-STL]$ ./a.out
0 0 1 1 0
0 0 0 1 0
0 0 0 0 0
0 0 1 0 0
1 1 0 1 1
bool t = b.test(n);   t= 0
b.size()=5
b.count()=4
b1!=b2
0 0 0 0 0
0 0 0 0 0
0 1 0 0 0
0 0 0 0 0
0 1 0 0 0

```

```
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

```
*/
```

Sentenças para bitset

- O valor default de cada bit é 0.
- Um bitset pode ser construído a partir de uma string de 0 e 1.
- Todo acesso `b[i]` é verificado, se `i` esta fora do intervalo, um `out_of_range` é disparado.
- Não confunda um operador sobre bits (`&` e `|`) com operadores lógicos (`&&` e `||`).

Listing 27.2: Usando bitset - Exemplo 2

```
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include <iomanip>
using std::setw;

#include <bitset>
#include <cmath>

intmain ()
{
    const int    dimensao =    10;
    std::bitset < dimensao > b;
    cout << b << endl;
    b.flip ();
    cout << b << endl;

    cout << " 0 bitset tem a dimensao:" << b.size () << endl;
    cout << " Entre com a posição do bit que quer inverter (0->9):";
    int    pos;
    cin >> pos;
    cin.get ();
    b.flip (pos);
    cout << b << endl;

    return 1;
}

/*
Saída:
-----
[andre@mercurio Cap4-STL]$ ./a.out
0000000000
1111111111
```

```

0 bitset tem a dimensao:10
Entre com a posição do bit que quer inverter (0->9):5
1111011111

*/

```

Listing 27.3: Usando bitset com vector

```

/*
Uso de vector e bitset
*/

#include<bitset>
#include<vector>
using namespace std;

void main ()
{
    //dimensao fixa do vetor de bits
    const int dimensao = 24;

    //dimensao variavel do vetor de celulas
    cout << "Entre com o numero de celulas:";
    int ncelulas;
    cin >> ncelulas;
    cin.get ();

    vector < bitset < dimensao > >v (ncelulas);

    //v[i] acessa o vetor i
    //pode usar set(0) a set(23)
    for (int i = 0; i < v.size (); i++)
        for (int j = 0; j < dimensao; j++)
            v[i][j] = ((i * j) % 2);

    for (int i = 0; i < v.size (); i++)
    {
        cout << endl;
        for (int j = 0; j < dimensao; j++)
            cout << v[i][j];
    }
    cout << endl;

    int d;
    do
    {
        int x, y;
        cout << "Entre com a celula que quer alterar 0->" << v.size () -
            1 << ": ";
        cin >> x;
        cout << "Entre com o bit que quer alterar de 0->" << dimensao << ": ";
        cin >> y;
        cout << "Entre com o novo valor do bit (0 ou 1), >1 encerra programa: ";
        cin >> d;
        cin.get ();
    }
}

```

```
v[x][y] = d;
cout << "\ncelula_" << x << "_bit_" << y << "_valor_" << v[x][y] <<
endl;
}
while (d <= 1);
}

/*
Como usa o vector,
//cria vetor de 50 inteiros
vector<int> v(50);
//cria vetor de 50 objetos do tipo bitset<dimensao>
vector<bitset<dimensao> > v(50);
//com v[i] acessa cada objeto bitset
com v[i][j] acessa objeto bitset i posicao j do bitset.
*/

/*
Novidade:
Uso de vector
Uso de bitset
*/

/*
Saída:
[root@mercurio Cap4-STL]# ./a.out
Entre com o numero de celulas : 3

00000000000000000000000000000000
010101010101010101010101010101
00000000000000000000000000000000
Entre com a celula que quer alterar 0->2: 1
Entre com o bit que quer alterar de 0->24: 2
Entre com o novo valor do bit(0 ou 1),>1 encerra programa: 1
celula 1 bit 2 valor =1
*/
```

Parte IV
Introdução a STL

Capítulo 28

Introdução a Biblioteca Padrão de C++ (STL)

No capítulo de Tipos, você aprendeu que C++ é fundamentada no uso de tipos, e que existem 3 tipos fundamentais: os tipos básicos de C++ (char, int, double,...), os tipos definidos pelo usuário (TPonto, TCirculo,...) e os tipos definidos em bibliotecas externas. Como exemplos de bibliotecas já abordamos o uso de strings, de números complexos e da hierarquia de entrada e saída de dados. Neste capítulo descreve-se a biblioteca padrão de C++, a standart template library (ou STL).

Descrever todas os conceitos e capacidades da STL é algo impossível de ser realizado em poucas páginas. O procedimento a ser adotado nesta apostila requer *atenção redobrada* por parte do aprendiz.

Inicia-se descrevendo o que é a STL, suas características e componentes. A seguir apresenta-se os diferentes tipos de containers e os seus métodos mais usuais. Segue-se descrevendo os iteradores, seus tipos e operações.

Parte-se então para a descrição prática de cada container da STL, iniciando-se com <vector> que é descrita em detalhes e a seguir os demais containers <list> <deque> <set> <multiset> <map> <multimap> <stack> <queue>.

28.1 O que é a STL?

A STL ou *Standart Template Library* é uma biblioteca de objetos avançada e útil. A mesma foi construída por Alexander Stepanov, Meng Lee, David Musser, usando os conceitos mais modernos da programação orientada a objeto. Todo desenvolvimento da STL foi acompanhado e aprovado pelo comite standart do C++ (o ANSI C++).

28.1.1 Características da STL:

- Não usa polimorfismo em função do desempenho.
- Usa extensivamente os templates.
- Construída basicamente sobre três conceitos: container's, iteradores e código genérico.

28.1.2 Componentes da STL

A STL é construída sobre os containers os iteradores e código genérico.

Containers: Primeiro vamos descrever o que é um container, quais os tipos de container e seus usos mais comuns. Inclue os container's de sequência, os associativos e os adaptativos. A seguir descreve-se os métodos e typedef's que são comuns entre os container's, ou seja, os conceitos válidos para todos os container's.

Iteradores: A seguir, descreve-se os iteradores, o que são, quais os tipos e as características dos diferentes tipos de iteradores. A seguir descreve-se os operadores comuns a todos os iteradores.

Código Genérico: Descreve-se então o uso das funções genéricas. Métodos de uso comum a adaptadas para ter um funcionamento muito íntimo com os containers da STL.

28.2 Introdução aos containers

Se você tem um grupo de objetos do mesmo tipo, você pode organizá-los através de um container.

Existem diferentes tipos de containers e a seleção do container mais adequado, vai depender do que você pretende realizar sobre o seu grupo de objetos. Segue abaixo uma lista dos diferentes tipos de containers e suas principais características.

28.2.1 Tipos de Container's

Os containers podem ser divididos em 3 categorias, os sequenciais, os associativos e os adaptativos. Descreve-se o nome do container, suas características básicas e os iteradores suportados.

Container's sequências:

Os containers sequências são vector, list e deque. Os mesmos são brevemente descritos a seguir.

vector:

Funciona como um vetor comum permitindo acesso aleatório.

Tem rápida inserção de objetos no final do vetor e lenta no meio.

```
# include <vector>
```

Iterador suportado: random acess.

list:

Use list se precisar de uma lista de objetos em que novos objetos podem ser incluídos em qualquer posição, ou seja, tem inserção e deleção rápida em qualquer posição.

Lento para acesso randômico.

```
# include <list>
```

Iterador suportado: bidirecional.

deque:

Use se precisar de uma lista de objetos, em que novos objetos podem ser incluídos em qualquer posição. Tem vantagens de vector e list. É uma fila com duas pontas.

Permite acesso aleatório.
include <deque>
Iterador suportado: random access.

Container's Associativos:

Os container's associativos funcionam com o conceito de chaves (keys). Os containers associativos estão sempre ordenados, por default o operador < é usado para ordenação.

set:
Um set armazena um conjunto de chaves (sem repetições).
include <set>
Iterador suportado: bidirecional.

multiset:
Um multiset armazena um conjunto de chaves (com repetições).
include <set>
Iterador suportado: bidirecional.

map:
Um map armazena um conjunto de pares [chave,objeto] (sem repetições).
include <map>
Iterador suportado: bidirecional.

multimap:
Um multimap armazena um conjunto de pares [chave,objeto] (com repetições).
include <map>
Iterador suportado: bidirecional.

Container's adaptativos:

São container's criados a partir da adaptação de um container de sequência, ou seja, pode ser construído tendo como base um vector ou um list ou um deque.

stack:
Um container que funciona como uma pilha LIFO (last in, first out)(o último que entra é o primeiro que sai). Semelhante a pilha de uma calculadora HP. Pode ser construído tendo como base um vector, list (default), deque.
include <stack>
Iterador suportado: não suportado.

queue:
Um container que funciona como uma fila FIFO (first in, first out)(o primeiro que entra é o primeiro que sai). Pode ser construído tendo como base um vector, list(default), ou deque.
include <queue>
Iterador suportado: não suportado.

priority_queue:

Um container que funciona como uma fila ordenada, onde quem sai é sempre o maior valor. Os elementos estão sempre ordenados. Pode ser construído tendo como base um list ou um deque (default).

include <priority_queue>

Iterador suportado: não suportado.

28.2.2 Métodos comuns aos diversos container's

Algumas funções e operadores estão presentes em todos os container's, estas funções e operadores são listados abaixo.

contrutor default: Cada container tem um conjunto de construtores válidos.

contrutor de cópia: Cria um container novo, uma cópia de um existente.

destrutor: Destrói o container.

empty(): Retorna true se o container esta vazio.

max_size(): Retorna o número máximo de elementos do container (valor alocado).

size(): Retorna o número de elementos usados.

operator=: Atribue os elementos de um container a outro.

operator<: C_A < C_B

Retorna true se C_A é menor que C_B

C_A é o container A e C_B o container B.

operator<=: C_A <= C_B

Retorna true se C_A é menor ou igual a C_B.

operator>: C_A > C_B

Retorna true se C_A é maior que C_B.

operator>=: C_A >= C_B

Retorna true se C_A é maior ou igual a C_B.

operator==: C_A == C_B

Retorna true se C_A é igual a C_B.

operator!=: C_A != C_B

Retorna true se C_A é diferente de C_B.

swap: Troca todos os elementos do container.

Métodos válidos apenas para os container's sequenciais (vector, list, deque)

Alguns métodos são válidos apenas para os containers sequenciais, isto é, para vector, deque e list. Estes métodos são listados a seguir. A Tabela 28.1 mostra alguns iteradores e a posição para onde apontam.

begin: Retorna um iterador (iterator ou const_iterator) para o primeiro elemento do container (posição 0).

end: Retorna um iterador (iterator ou const_iterator) para o último elemento do container (elemento não utilizado), (posição n).

rbegin: Retorna um iterador (iterator ou const_iterator) para o último elemento do container (posição n-1).

rend: Retorna um iterador (iterator ou const_iterator) para o elemento anterior ao primeiro objeto do container (elemento não utilizado) (posição -1).

erase: Apaga um ou mais elementos do container.

clear: Apaga todos os objetos do container.

Tabela 28.1: Iteradores e posição.

<i>rend</i>	<i>begin</i>	<i>..</i>	<i>rbegin</i>	<i>end</i>
<i>-1</i>	<i>0</i>		<i>n-1</i>	<i>n</i>

A Figura 28.1 mostra um diagrama mostrando os métodos comuns aos diversos containers. Observe a presença de métodos para inclusão e eliminação de objetos do container, métodos que retornam iteradores para os objetos do container e métodos que retornam a dimensão e capacidade do container. Os métodos `push_front` e `push_back` são utilizados para adicionar ao container objetos no início e no fim do container, respectivamente. Os métodos `pop_front`, `erase` e `pop_back` são utilizados para deletar objetos do container. Você pode obter cópias dos objetos utilizando `front`, `at`, e `back`. Para verificar o tamanho alocado do container use `capacity`, para obter o número de elementos utilizados use `size` e para obter o limite máximo que o container pode ter use `max_size`.

28.2.3 Typedef's para containers²

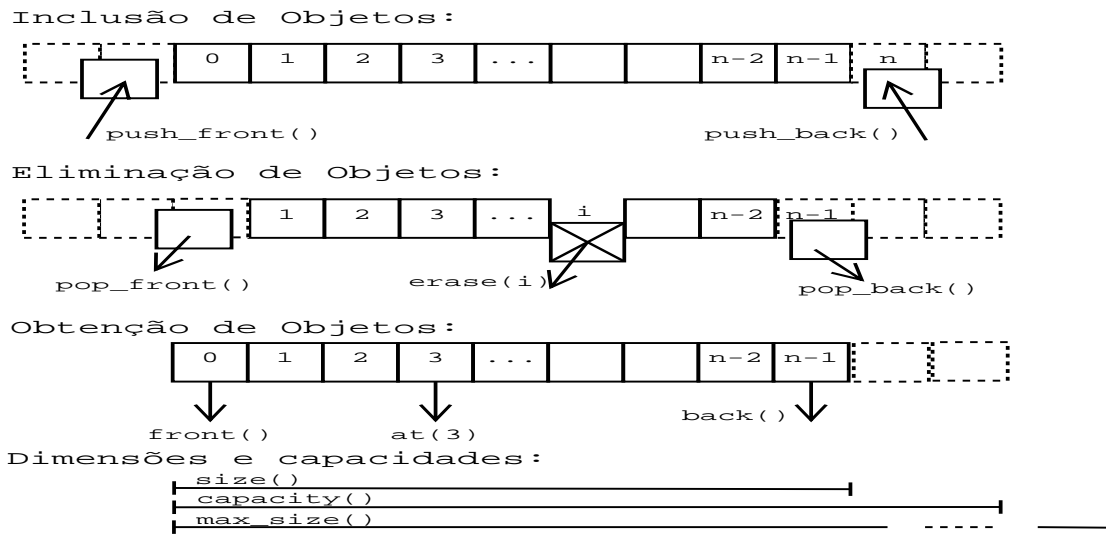
Lembre-se um typedef nada mais é do que um apelido para uma declaração de um objeto.

Exemplo:

```
typedef float racional;
typedef const float cfloat;
typedef set<double, less<double> > double_set;
```

Alguns typedef's estão presentes em todos os container's, estes typedef's são listados abaixo.

Figura 28.1: Métodos comuns aos diversos containers.



value_type: O tipo de elemento armazenado no container.

size_type: Tipo usado para contar itens no container e indexar uma sequência de container's. Inválido para list.

reference: Uma referência para o tipo armazenado no container.

pointer: Um ponteiro para o tipo armazenado no container.

iterator: Um iterador para o tipo armazenado no container.

reverse_iterator: Um iterador reverso para o tipo armazenado no container.

allocator_type: Tipo de gerenciamento de memória utilizado.

difference_type: Número de elementos entre dois iteradores. Não definido para os container's list e adaptativos (stack, queue, priority_queue).

const_pointer: Um ponteiro constante para o tipo armazenado no container.

const_iterator: Um iterador constante para o tipo armazenado no container.

const_reverse_iterator: Um iterador reverso constante para o tipo armazenado no container.

Os métodos begin, end, rbegin e rend retornam iteradores, veremos a seguir que iteradores são objetos ponteiros.

28.3 Introdução aos iteradores (iterator's)

O que é um iterador?

Um iterador é um ponteiro inteligente é um objeto ponteiro. Os iteradores foram desenvolvidos para dar suporte aos container's já descritos. Lembre-se, um ponteiro aponta para determinada posição da memória e você pode se deslocar por um vetor de objetos usando o ponteiro, com um iterador você faz a mesma coisa.

Descreve-se a seguir os diferentes tipos de iteradores, os typedefs padrões e os operadores que são sobrecarregados para os iteradores.

28.3.1 Tipos de iteradores

Existe uma certa hierarquia entre os iteradores, os dois mais simples são o input e o output, pois permitem apenas operações de leitura e escrita (respectivamente). A seguir vem o forward, que permite leitura e escrita (mas somente para frente). O bidirecional permite leitura e escrita tanto para frente quanto para trás. O mais poderoso é o randomico que permite a leitura e escrita randomicamente.

Sequência: input,output->forward->bidirecional->random access.

Lista-se a seguir as características dos iteradores:

input: Lê um objeto do container, se move do início para o fim do container.
algorithm: suporta somente uma passagem.

output: Escreve um objeto no container, se move do início para o fim do container.
algorithm: suporta somente uma passagem.

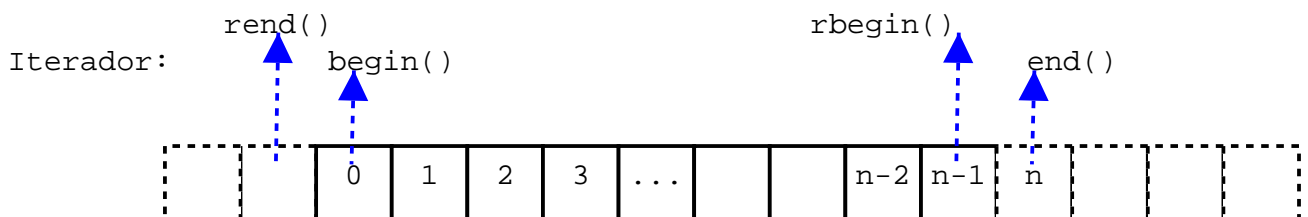
forward: Leitura e escrita somente para frente.

bidirecional: Leitura e escrita para frente e para trás.

random access: Leitura e escrita acessando randomicamente qualquer objeto do container.

A Figura 28.2 mostra um diagrama mostrando como obter os iteradores de um container.

Figura 28.2: Métodos que retornam iteradores.



28.3.2 Operações comuns com iteradores²

Algumas operações comuns aos diferentes tipos de iteradores são listadas abaixo.

Iteradores de leitura (input)

`++p`: Pré-incremento.

`p++`: Pos-incremento.

`*p`: Retorna objeto (desreferencia ponteiro).

`p=p1`: Atribue um iterador a outro.

`p==p1`: Compara se dois iteradores são iguais.

`p!=p1`: Compara se dois iteradores são diferentes.

Iteradores de escrita (output)

`++p`: Pré-incremento.

`p++`: Pos-incremento.

`*p`: Retorna objeto (desreferencia ponteiro).

`p=p1`: Atribue um iterador a outro.

Iteradores de avanço (forward)

`++p`: Pré-incremento.

`p++`: Pos-incremento.

`*p`: Retorna objeto (desreferencia ponteiro).

`p=p1`: Atribue um iterador a outro.

`p==p1`: Compara se dois iteradores são iguais.

`p!=p1`: Compara se dois iteradores são diferentes.

Iteradores Bidirecionais

`++p`: Pré-incremento.

`p++`: Pos-incremento.

`-p`: Pré-decremento.

`p-`: Pós-decremento.

Iteradores randomicos

- ++p: Pré-incremento.
 - p++: Pos-incremento.
 - p+=i: iterador avança i posições.
 - p-=i: iterador recua i posições.
 - p+i: Retorna iterador avançado i posições de p.
 - p-i: Retorna iterador recuado i posições de p.
 - p[i]: Retorna referência ao objeto i.
 - p<p1: True se p aponta para elemento anterior a p1.
 - p<=p1: True se p aponta p/elemento anterior/igual a p1.
 - p>p1: True se p aponta para elemento acima de p1.
 - p>=p1: True se p aponta para elemento acima/igual a p1.
- O uso dos iteradores serão esclarecidos através dos exemplos.

Capítulo 29

class <vector>

O container `vector` vai ser apresentado em detalhes. Os demais containers serão apresentados de uma forma simplificada, pois as suas características são semelhantes a `vector`. Isto significa, que você deve ler este capítulo com atenção redobrada, pois seus conceitos se aplicam aos demais containers.

Como o conjunto de funções fornecidas por cada container é grande, não tente decorar nada, apenas preste atenção na idéia. Com os exemplos você irá aprender a usar as classes container com facilidade.

- O container `vector` funciona como um vetor comum, ou seja, os blocos de objetos estão contíguos, permitindo acesso aleatório.
- Como em vetores comuns de C, `vector` não verifica os índices.
- `vector` permite iteradores randomicos.
- É um container sequencial.

Para usar um objeto container do tipo `vector` inclua o header <vector>

```
Exemplo:  
# include <vector>
```

Apresenta-se a seguir os diversos métodos disponibilizados por <vector>, primeiro apresenta-se os construtores e depois os métodos usuais de `vector`.

Construtores e Destrutores

Cria um `vector` com tamanho zero.

```
vector ();
```

```
vector<int> v_int(15);
```

Cria um `vector` com tamanho `n`, com `n` cópias do valor.

```
vector (size_type n, const T& valor = T());
```

```
vector<float> v_float(15,3.55);
```

Cria um vector do tamanho de last-first, com os valores de first;

```
template <class InputIterator>vector (InputIterator first, InputIterator last);
```

```
vector<float> v_float2 (v_float.begin(), v_float.end());
```

Construtor de cópia, cria uma cópia do vector v.

```
vector (const vector<T>& v);
```

```
vector<float> v_float3 (v_float);
```

Destrutor de vector

```
~vector ();
```

Iteradores de vector

Retorna um iterador randomico para o primeiro elemento.

```
iterator begin ();
```

Retorna um iterador randomico constante para o primeiro elemento.

```
const_iterator begin () const;
```

Retorna um iterador randomico para o último elemento (posição n).

```
iterator end ();
```

Retorna um iterador randomico constante para o último elemento.

```
const_iterator end () const;
```

Retorna um iterador randomico reverso para o último elemento válido (posição n-1).

```
reverse_iterator rbegin ();
```

Retorna um iterador randomico reverso constante para o último elemento válido (posição n-1).

```
const_reverse_iterator rbegin () const;
```

Retorna um iterador randomico para o primeiro elemento.

```
reverse_iterator rend ();
```

Retorna um iterador randomico constante para o primeiro elemento.

```
const_reverse_iterator rend () const;
```

Referências e acesso

Retorna uma referência ao primeiro elemento.

reference front ();

Retorna uma referência constante ao primeiro elemento.

const_reference front () const;

Retorna uma referência ao objeto *n* (não verifica o intervalo). Só é válido para `vector` e `deque`.

operator[] (size_type n);

Retorna uma referência constante ao objeto *n*.

const_reference operator[] (size_type n) const;

Retorna uma referência ao objeto *n*, at testa o intervalo.

at(size_type n);

Retorna uma referência constante ao objeto *n*.

const_reference at (size_type) const;

Retorna uma referência ao último elemento.

reference back ();

Retorna uma referência constante ao último elemento.

const_reference back () const;

Operadores

Operador de atribuição. Apaga todos os elementos e depois copia os valores de *x* para o container. Retorna uma referência para o conjunto.

vector<T>& operator= (const vector<T>& x);

Capacidade e redimensionamento

Retorna o tamanho alocado do container (memória alocada).

size_type capacity () const;

Retorna true se o container esta vazio.

bool empty () const;

Retorna o tamanho do maior vetor possível.

size_type max_size () const;

Define a capacidade do container em `n` elementos, útil quando você sabe que vai adicionar um determinado número de elementos ao container, pois evita a realocação do container a cada nova inclusão. Se `n` for menor que a capacidade atual do container a realocação não é realizada.

```
void reserve (size_type n);
```

Altera o tamanho do container.

Se o novo tamanho (`n`) for maior que o atual, os objetos `c` são inseridos no final do vetor.

Se o novo tamanho for menor que o atual, os elementos excedentes são eliminados.

```
void resize (size_type sz, T c = T());
```

Retorna o número de elementos do container.

```
size_type size () const;
```

Inserção, deleção e atribuição

Inserir o objeto `x`, antes da posição definida pelo iterador.

```
iterator insert (iterator position, const T& x = T());
```

Inserir `n` cópias de `x` antes da posição.

```
void insert (iterator position, size_type n, const T& x = T());
```

Inserir cópia dos elementos no intervalo `[first, last]` antes da posição `p`.

```
template <class InputIterator>
```

```
void insert (iterator p, InputIterator first, InputIterator last);
```

Inserir uma cópia de `x` no final do container.

```
void push_back (const T& x);
```

Remover o elemento da posição `p`.

```
void erase (iterator p);
```

```
    erase (it+int);
```

Remover os elementos no intervalo (inclusive `first`, excluindo `last`), ou seja, de `first` a `last-1`.

```
void erase (iterator first, iterator last);
```

Remover o último elemento (sem retorno).

```
void pop_back ();
```

Apagar todos os elementos

```
void clear();
```

Apagar todos os elementos do container e inserir os novos elementos do intervalo `[first, last)`.

```
template <class InputIterator> void assign (InputIterator first, InputIterator last);
```

Apaga todos os elementos do container e insere os n novos elementos com o valor de t .
template <class Size, class T>void assign (Size n, const T& t = T());

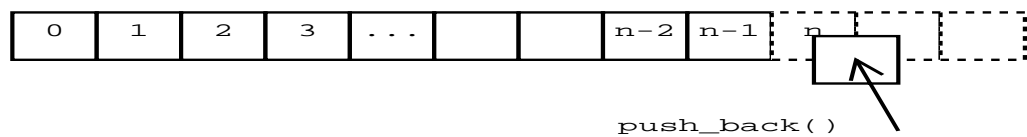
Troca os elementos x (é mais rápida que a swap genérica).

void swap (vector<T>& x);

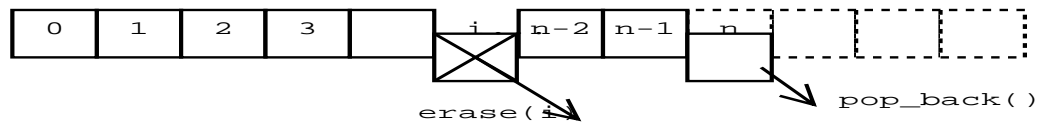
A Figura 29.1 mostra um diagrama mostrando os métodos para inserção e deleção de objetos em um container. Observe que é um diagrama genérico, alguns de seus métodos não se aplicam a vector, como `push_front` e `pop_front`.

Figura 29.1: Métodos disponibilizados para vector.

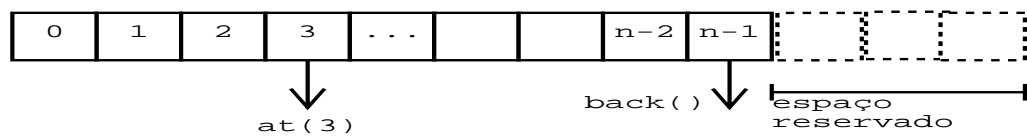
Inclusão de Objetos:



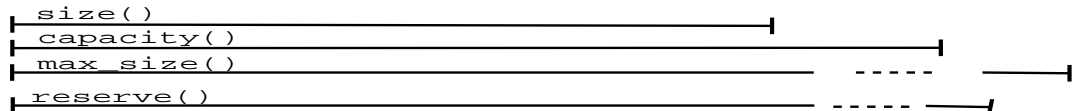
Eliminação de Objetos:



Obtenção de Objetos:



Dimensões e capacidades:



Operadores não membros

Retorna true se x é igual a y (se cada elemento é igual).

template <class T>

bool operator== (const vector<T>& x, const vector <T>& y);

Retorna true se os elementos contidos em x são "lexicographically" menores que os elementos contidos em y .

template <class T>bool operator< (const vector<T>&x,const vector<T>&y);

Os outros operadores sobrecarregados são:

`!=`, `<=`, `>=`.

29.1 Sentenças para vector

- Quando possível, use a biblioteca padrão.
- Se não for mudar o container use iteradores do tipo `const`.
- Sempre reserve um espaço inicial para o vetor, ou seja, procure evitar a realocação a cada inserção, alocando todo o bloco que vai utilizar de uma única vez.
- Os métodos devem receber vector como referência para evitar cópias desnecessárias.

Exemplo:

```
void funcao(vector<int> & );
```

- Se a classe não tem um construtor default, um vector só pode ser criado passando-se o construtor com parâmetros.

Exemplo:

```
class TUsuario{TUsuario(int x){...}};  
vector<TUsuario> vet(200,TUsuario(5));
```

- Os métodos `assign` são usadas complementarmente aos construtores. O número de elementos do vetor serão aqueles passados para `assign`.
- Se o método espera um iterador você não deve passar um `reverse-iterator`.
- Para converter um `reverse_iterator` em um iterador use a função `base()`.

Exemplo:

```
reverse_iterator ri...;  
iterator it = ri.base();
```

- Se for inserir objetos no meio do vetor, pense em usar uma lista (`list`).
- `vector` não possui `push_front`, `pop_front`, `front`.
- Os container's já vem com os operadores `<` e `==`. Você pode sobrecarregar `x > y`, `!=`, `<=` e `>=`. Ou usar:

```
#include <utility>  
using namespace std::rel_ops;
```

Os exemplos a seguir mostram o uso de vector. Você pode rever o exemplo de vector visto no Capítulo Tipos.

Listing 29.1: Usando vector

```
//Classes para entrada e saída
#include <iostream>

//Classe pra formatação de entrada e saída
#include <iomanip>

//Classe de vetores, do container vector
#include <vector>

//Classe para algoritimos genéricos
//#include <algorithm>

//Define estar usando espaço de nomes std
using namespace std;

//Sobrecarga do operador <<
ostream & operator<< (ostream & os, const vector < int >&v);

//Definição da função main
int main ()
{
    //Cria vector, do tipo int, com nome v
    vector < int >v;

    int data;
    cout << "No_DOS_um_ctrl+z_encerra_a_entrada_de_dados." << endl;
    cout << "No_Mac_um_ctrl+d_encerra_a_entrada_de_dados." << endl;
    cout << "No_Linux_um_ctrl+d_encerra_a_entrada_de_dados." << endl;
    do
    {
        cout << "\nEntre_com_o_dado(" << setw (3) << v.size () << "):";
        cin >> data;
        cin.get ();
        if(cin.good ())
            v.push_back (data);
    }
    while (cin.good ());

    cout << "\n";
    cout << v << endl;

    //Alterando diretamente os elementos do vetor
    v[0] = 23427;
    //v.at( 1 ) = 13120;
    //inserindo na posição 2
    v.insert (v.begin () + 2, 5463);
    cout << "\nApós_v[0]=23427;e_v.insert(v.begin()+2,5463);" << endl;
    cout << v << endl;

    //Chama função erase do objeto vector passando posição v.begin
    v.erase (v.begin ());
    cout << "\nApós_v.erase(v.begin());" << endl;
    cout << v << endl;
}
```

```

//Chama função erase do objeto vector passando v.begin e v.end
cout << "\nApós v.erase(v.begin(), v.end());" << endl;
v.erase (v.begin (), v.end ());
cout << "o vetor esta" << (v.empty ())? "vazio" : "com elementos" << endl;
cout << v << endl;

//Chama função clear
v.clear ();
cout << "o vetor esta" << (v.empty ())? "vazio" : "com elementos" << endl;

cout << endl;
cin.get ();
return 0;
}

//Uso de sobrecarga do operador << para mostrar dados do vetor.
ostream & operator<< (ostream & os, const vector < int >&v)
{
    for (int i = 0; i < v.size (); i++)
        {
            os << "v[" << setw (3) << i << "]= " << setw (5) << v[i] << ' ';
        }
    return os;
}

/*
Novidades:
Uso sobrecarga do operador << para mostrar dados do vetor.
Uso de insert e erase.
*/

/*
Saída:
-----
[andre@mercurio Cap4-STL]$ g++ ex-vector-2.cpp
[andre@mercurio Cap4-STL]$ ./a.out
No DOS    um ctrl+z encerra a entrada de dados.
No Mac    um ctrl+d encerra a entrada de dados.
No Linux  um ctrl+d encerra a entrada de dados.

Entre com o dado ( 0):0
Entre com o dado ( 1):-1
Entre com o dado ( 2):-2
Entre com o dado ( 3):-3
Entre com o dado ( 4):

v[ 0]=    0 v[ 1]=   -1 v[ 2]=   -2 v[ 3]=   -3

Após v[ 0 ] = 23427; e v.insert( v.begin() + 2, 5463 );
v[ 0]=23427 v[ 1]=   -1 v[ 2]= 5463 v[ 3]=   -2 v[ 4]=   -3

Após v.erase( v.begin() );
v[ 0]=   -1 v[ 1]= 5463 v[ 2]=   -2 v[ 3]=   -3

Após v.erase( v.begin(), v.end() );

```

```
o vetor esta vazio  
o vetor esta vazio  
*/
```


Capítulo 30

class <list>

Use list se precisar de uma lista de objetos em que novos objetos podem ser incluídos em qualquer posição, ou seja, tem inserção e deleção rápida em qualquer posição.

- Um container list é lento para acesso randômico.
- É otimizado para inserção e remoção de elementos.
- list suporta iterador bidirecional.
- Um list fornece todas as operações de um vector, com exceção de at[], capacity() e reserve().
- list tem métodos novos, como, front, push_front e pop_front.
- É um container sequencial.
- Para usar list inclua o header

```
# include <list>
```

A Figura 30.1 mostra os métodos disponibilizados para list.

Construtores e destrutores

Cria uma lista com zero elementos.

```
list();
```

Cria lista com tamanho n, com n copias do valor.

```
list (size_type n, const T& value = T());
```

Cria uma lista do tamanho de last-first, com os valores de first.

```
template <class InputIterator>list (InputIterator first, InputIterator last);
```

Cria uma cópia da lista v.

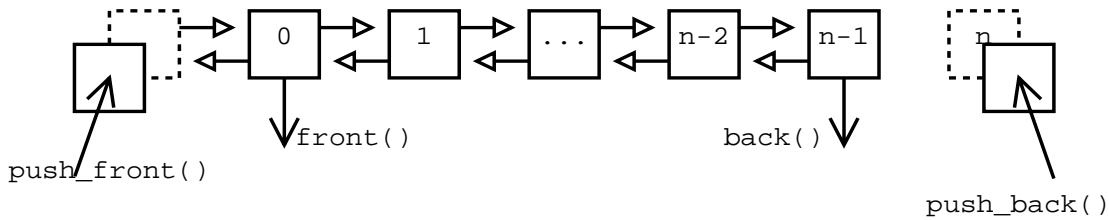
```
list (const list<T>& x);
```

Destrutor de list.

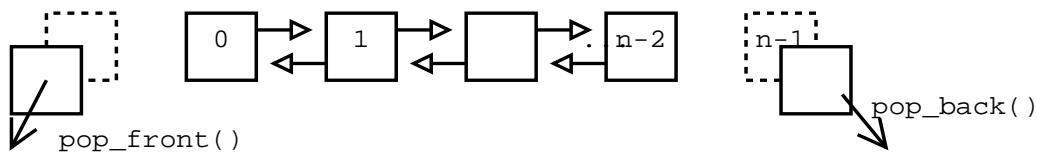
```
~list ();
```

Figura 30.1: Métodos disponibilizados para list.

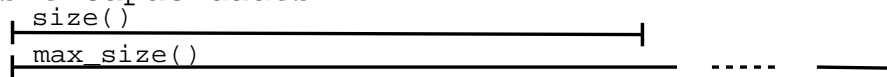
Inclusão e obtenção de Objetos:



Eliminação de Objetos:



Dimensões e capacidades:



Operadores

Operador de atribuição. Apaga todos os elementos e depois copia os valores de x para o container. Retorna uma referência para o conjunto.

```
list<T>& operator= (const list<T>& x)
```

Métodos front/push_front/pop_front

Referência ao primeiro elemento.

```
reference front();
```

Remove o primeiro elemento.

```
void pop_front ();
```

Adiciona uma cópia de x no início da lista (novo primeiro elemento).

```
push_front (const T& x);
```

Métodos splice/merge/sort

Insere x antes da posição definida pelo iterator.

```
void splice (iterator position, list<T>& x);
```

Move de list[i] para this[position]. list[i] é uma outra lista recebida como parâmetro, this é a própria lista.

```
void splice (iterator position, list<T>& x, iterator i);
```

Move de list[first,last] para this[position].

Move os elementos no intervalo [first, last] da lista x para this, inserindo antes de position.

```
void splice (iterator position, list<T>& x, iterator first, iterator last);
```

Mistura valores de x ordenados com o this, usando operador<. Se existirem elementos iguais nas duas listas, os elementos de this precedem, a lista x ficará vazia.

```
void merge (list<T>& x);
```

Mistura x com this, usando função de comparação. Se existirem elementos iguais nas duas listas, os elementos de this precedem, a lista x ficará vazia.

```
template <class Compare> void merge (list<T>& x, Compare comp);
```

Ordena de acordo com o operador <. Elementos iguais são mantidos na mesma ordem.

```
void sort ();
```

Ordena a lista de acordo com a função de comparação.

```
template <class Compare> void sort (Compare comp);
```

Métodos unique e remove

.

Move todos os elementos repetidos para o fim do container e seta size como sendo o índice do último elemento não duplicado. Antes de executar unique execute um sort.

```
void unique ();
```

Apaga elementos consecutivos com a condição true dada pelo binary_pred. A primeira ocorrência não é eliminada.

```
template <class BinaryPredicate> void unique (BinaryPredicate binary_pred);
```

Remove os elementos com valor val.

```
void remove(const T& val);
```

Remove os elementos que satisfazem a condição predicado p.

```
template <class pred> void remove_if(Pred p);
```

Inverte a ordem dos elementos.

```
void reverse();
```

30.1 Sentenças para list

- Uma lista não aceita (iterator + int). Você precisa fazer iterator++ n vezes.

- Objetos eliminados de list são deletados.
- list não aceita subscript[], reserve() e capacity.
- list inclui splice(pos,& x); move x para pos.
- Use list sempre que precisar de inserção e remoção rápida.
- merge(list &); mescla listas.

Exemplo:

```
//remove da lista os objetos que tem a primeira letra =='p'.
list.remove_if(initial('p'));
```

Veja a seguir um exemplo de uso de list. Observe na função de sobrecarga de operador o uso do operador de extração com um iterator.

Listing 30.1: Usando list.

```
#include <iostream>
#include <string>

//Classe de listas
#include <list>

//Algoritmo genérico
#include <algorithm>

using namespace std;

//Sobrecarga operador extração << para list
ostream & operator<< (ostream & os, const std::list < float >&lista);

//Definição da função main
int
main ()
{
    string linha = "\n-----\n";

    //Criação de duas listas para float
    std::list < float >container_list, container_list2;

    //Inclue valores na lista
    container_list.push_front (312.1f);
    container_list.push_back (313.4f);
    container_list.push_back (316.7f);
    container_list.push_front (312.1f);
    container_list.push_front (313.4f);
    container_list.push_front (314.1f);
    container_list.push_front (315.1f);
    cout << linha << "Conteúdo do container:" << endl;
    cout << container_list << linha << endl;
```



```

//elimina primeiro elemento da lista
container_list.pop_front ();
cout << "Conteúdo do container após: " << container_list.pop_front(); << endl;
cout << container_list << linha << endl;

//elimina ultimo elemento da lista
container_list.pop_back ();
cout << "Conteúdo do container após: " << container_list.pop_back(); << endl;
cout << container_list << linha << endl;

//ordena o container
container_list.sort ();
cout << "Conteúdo do container após: " << container_list.sort(); << endl;
cout << container_list << linha << endl;

//move os elementos repetidos para o final do container
//e seta como último elemento válido, o último elemento não repetido.
//
container_list.unique ();
cout << "Conteúdo do container após: " << container_list.unique(); << endl;
cout << container_list << linha << endl;

cin.get ();
return 0;
}

//Mostra lista
//com vector foi possível usar v[i], uma lista não aceita l[i],
//precisa de um iterator, como abaixo.
ostream & operator<< (ostream & os, const std::list < float >&lista)
{
    std::list < float >::const_iterator it;
    for (it = lista.begin (); it != lista.end (); it++)
        os << *it << ' ';
    return os;
}

/*
Novidades:
Uso de list.
Uso de push_front e push_back.
Uso de pop_front e pop_back.
Uso de sort, unique
Uso de const iterator
Uso de sobrecarga para <<
*/

/*
Saída:
-----
[root@mercurio Cap4-STL]# ./a.out
-----
Conteúdo do container:
315.1 314.1 313.4 312.1 312.1 313.4 316.7

```

```
-----  
Conteúdo do container após: container_list.pop_front();  
314.1 313.4 312.1 312.1 313.4 316.7  
-----
```

```
Conteúdo do container após: container_list.pop_back();  
314.1 313.4 312.1 312.1 313.4  
-----
```

```
Conteúdo do container após: container_list.sort();  
312.1 312.1 313.4 313.4 314.1  
-----
```

```
Conteúdo do container após: container_list.unique();  
312.1 313.4 314.1  
-----
```

```
*/
```

Capítulo 31

class <deque>

Um container deque é uma fila com duas extremidades. Une parte das vantagens das lista (list) e dos vetores(vector). Tem rápida inserção na frente e atrás. De uma maneira geral são alocados blocos de memória que só são deletados quando o objeto é destruído.

- Tem os mesmos métodos de vector, mas adiciona `push_front` e `pop_front`.
- Classe container que suporta iteradores randomicos.
- É um container sequencial.
- Para usar um deque inclua

```
# include <deque>
```

A Figura 31.1 mostra os métodos disponibilizados para deque.

Construtores e destrutor

Construtor default, cria objeto vazio.

```
deque ();
```

Cria objeto com n elementos contendo n cópias de value.

```
deque (size_type n, const T& value = T());
```

Construtor de cópia.

```
deque (const deque<T>& x);
```

Cria um deque, do tamanho de last - first, preenchido com os valores de first a last

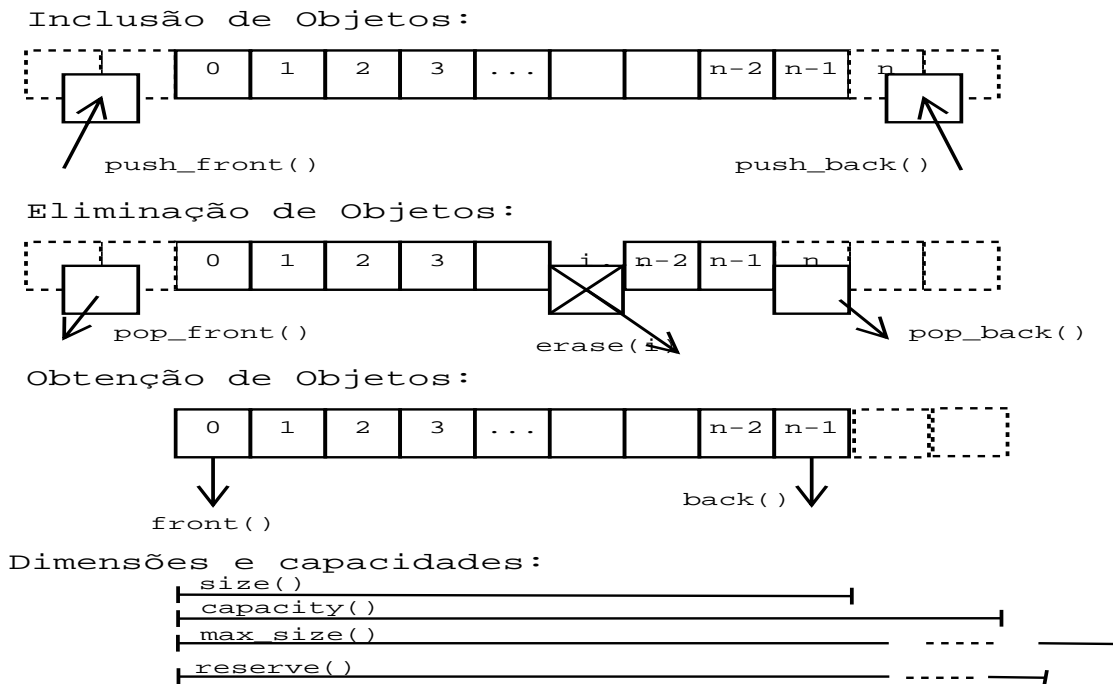
```
template <class InputIterator> deque (InputIterator first, InputIterator last);
```

Destrutor.

```
~ deque ();
```

Veja a seguir exemplo de uso de deque.

Figura 31.1: Métodos disponibilizados para deque.



Listing 31.1: Usando deque.

```
#include <iostream>

//Classe do container deque
#include <deque>

//Classe de algoritmo genérico
#include <algorithm>

//Declara o uso do namespace standart
using namespace std;

//Definição da função main
int main ()
{
    //Cria objeto do tipo deque para double com o nome container_deque
    deque < double >container_deque;

    //Adicionando objetos ao deque
    container_deque.push_front (1.3);        //adicionar no início
    container_deque.push_front (4.7);
    container_deque.push_back (4.5);        //adicionar no fim

    cout << "Conteúdo do container deque: ";

    for (int i = 0; i < container_deque.size (); ++i)
        cout << container_deque[i] << ' ';
}
```

```

//Retirando primeiro elemento do deque
container_deque.pop_front ();
cout << "\nApós um pop_front: ";
for (int i = 0; i < container_deque.size (); ++i)
    cout << container_deque[i] << ' ';

//Setando um objeto do container diretamente
container_deque[1] = 345.6;
cout << "\nApós atribuicao direta: container_deque[1] = 345.6;\n";
for (int i = 0; i < container_deque.size (); ++i)
    cout << container_deque[i] << ' ';

cout << endl;
return 0;
}

/*
Novidade:
Uso de deque (push_front e pop_front)
*/

/*
Saída:
-----
[root@mercurio Cap4-STL]# ./a.out
Conteúdo do container deque: 4.7 1.3 4.5
Após um pop_front: 1.3 4.5
Após atribuicao direta: container_deque[ 1 ] = 345.6;
1.3 345.6
*/

```


Capítulo 32

class <stack>

Um stack é um container adaptado que trabalha como uma pilha LIFO (last in, first out). O último elemento colocado na pilha é o primeiro a ser removido, da mesma forma que uma pilha de uma calculadora HP.

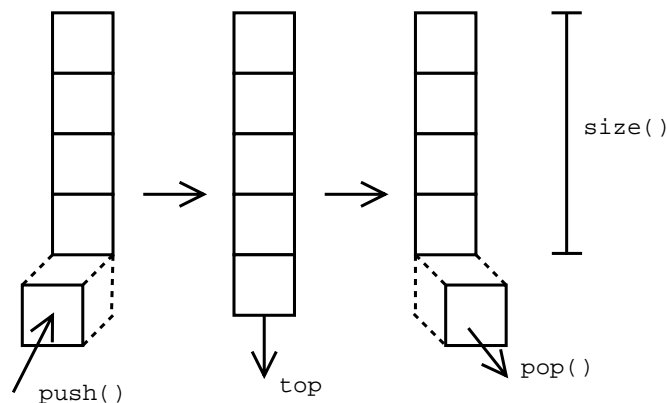
- Um container stack é um container adaptado porque o mesmo é construído sobre um container vector, list ou deque (default). Se não for fornecido um container, por default usa um deque.
- A classe stack é pequena e simples. Descreve-se a seguir toda a classe stack.
- Para usar um objeto stack inclua o header:

```
# include <stack>
```

A Figura 32.1 mostra os métodos disponibilizados para stack.

Figura 32.1: Métodos disponibilizados para stack.

Um stack funciona como uma pilha :



Construtores e typedefs de stack

```
template <class T, class Container = deque<T> >
class stack
{
typedef Container::value_type value_type;
typedef Container::size_type size_type;
protected:
Container c;
```

Métodos de stack

```
public:
//Coloca ítem x na pilha
void push (const value_type& x);

//Remove ítem da pilha
void pop ();

//Retorna o ítem no top da pilha, sem remover
value_type& top ();

//Retorna o ítem no topo da pilha, como valor constante.
const value_type& top () const;

//Retorna o número de elementos da pilha
size_type size () const;

//Retorna true se a pilha esta vazia
bool empty () ;
};
Apresenta-se a seguir um exemplo de uso do container stack.
```

Listing 32.1: Usando stack.

```
#include <iostream>
using std::cout;
using std::endl;

#include <stack>
#include <vector>
#include <list>

int main ()
{
//typedef
std::stack < int >
    container_stack_deque;
std::stack < int,
std::vector < int >>
    container_stack_vector;
```



```

std::stack < int,
std::list < int >>
    container_stack_list;

//Adicionando elementos ao container
for (int i = 0; i < 10; ++i)
{
    container_stack_deque.push (i);
    container_stack_vector.push (i * i);
    container_stack_list.push (i * i * i);
}

cout << "\nRetirando elementos do container_stack_deque: ";
while (!container_stack_deque.empty ())
{
    cout << container_stack_deque.top () << ' ';
    container_stack_deque.pop ();
}

cout << "\nRetirando elementos do container_stack_vector: ";
while (!container_stack_vector.empty ())
{
    cout << container_stack_vector.top () << ' ';
    container_stack_vector.pop ();
}

cout << "\nRetirando elementos do container_stack_list: ";
while (!container_stack_list.empty ())
{
    cout << container_stack_list.top () << ' ';
    container_stack_list.pop ();
}

cout << endl;
return 0;
}

/*
Novidade:
-----
Manipulacao de um container usando um stack.
Uso de push para adicionar elementos.
Uso de top para ver elemento topo da pilha.
Uso de pop para retirar elemento da pilha.
*/
/*
Saída:
-----
[root@mercurio Cap4-STL]# ./a.out

Retirando elementos do container_stack_deque:
9 8 7 6 5 4 3 2 1 0
Retirando elementos do container_stack_vector:
81 64 49 36 25 16 9 4 1 0
Retirando elementos do container_stack_list:

```

```
729 512 343 216 125 64 27 8 1 0
*/
```

Capítulo 33

class <queue>

Um container queue é um container que trabalha como se fosse uma fila do tipo FIFO (first in, first out).

- Os itens são adicionados (pushed) na parte de trás (back) e removidos da parte da frente (front).
- O tipo queue pode ser adaptado a qualquer container que suporte as operações front(), back(), push_back() and pop_front().
- Normalmente são usados com list e deque(default), não suporta vector.
- Para usar queue inclua o header

```
#include <queue>
```

Mostra-se a seguir a classe queue.

Construtores e Typedefs de queue

```
template <class T, class Container = deque<T> >
class queue
{
public:
typedef typename Container::value_type value_type;
typedef typename Container::size_type size_type;
protected:
    Container c;
public:
```

Métodos de queue

```
//Retorna o objeto do fim da lista (o último item colocado)
    value_type& back ();
```

```
//Retorna o objeto do fim da lista (o último item colocado) como const
```

```
const value_type& back() const;

//Retorna true se a fila esta vazia
bool empty () const;

//Retorna o ítem da frente da fila. É o primeiro ítem que foi
//colocado na fila (o primeiro que entra é o primeiro a sair)
value_type& front ();

//Retorna o ítem da frente da fila como const
const value_type& front () const;

//Remove o ítem da frente da fila
void pop ();

//Coloca x na parte de trás da fila
void push (const value_type& x);

//Retorna o número de elementos da fila
size_type size () const;
};
```

Capítulo 34

class <priority_queue>

Um container que funciona da mesma forma que um queue, a diferença é que com priority_queue a fila esta sempre ordenada, ou seja, o elemento retirado com pop é sempre o maior objeto.

- Um priority_queue compara seus elementos usando <, e então usa pop para retornar o maior valor.
- Para usar priority_queue inclua o header:

```
# include <queue>
```


Capítulo 35

class <set>

Trabalha com um conjunto de chaves. Num set as chaves não podem ser repetidas. Aceita acesso bidirecional, mas não randomico.

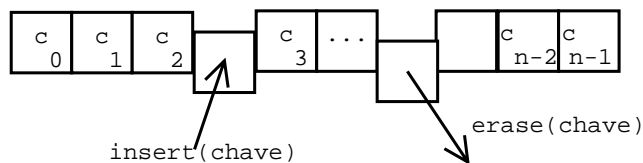
- É como um map, mas não armazena um valor, somente a chave.
- É um container associativo.
- Set define `value_type` como sendo a chave.
- Set inclui, `==`, `!=`, `<`, `>`, `<=`, `>=`, e `swap`.
- Para usar set inclua o header

```
# include <set>
```

A Figura 35.1 mostra os métodos disponibilizados para set.

Figura 35.1: Métodos disponibilizados para set.

Set armazena um conjunto de chaves:



typedefs para set

```
typedef Key key_type;  
typedef Key value_type;  
typedef Compare key_compare;  
typedef Compare value_compare;
```

Contrutores

```
set();  
explicit set(const Compare& comp);  
template <class InputIterator>  
set(InputIterator first, InputIterator last): t(Compare()) ;  
template <class InputIterator>  
set(InputIterator first, InputIterator last, const Compare& comp);  
set(const value_type* first, const value_type* last);  
set(const value_type* first, const value_type* last, const Compare& comp);  
set(const_iterator first, const_iterator last);  
set(const_iterator first, const_iterator last, const Compare& comp);  
set(const set<Key, Compare, Alloc>& x);  
set<Key, Compare, Alloc>& operator=(const set<Key, Compare, Alloc>& x)
```

Métodos de acesso

```
key_compare key_comp();  
value_compare value_comp();  
void swap(set<Key, Compare, Alloc>& x);
```

Métodos insert, erase e clear

```
typedef pair<iterator, bool> pair_iterator_bool;  
pair<iterator, bool> insert(const value_type& x) ;  
iterator insert(iterator position, const value_type& x) ;  
template <class InputIterator>  
void insert(InputIterator first, InputIterator last)  
void insert(const_iterator first, const_iterator last);  
void insert(const value_type* first, const value_type* last)  
void erase(iterator position) ;  
size_type erase(const key_type& x);  
void erase(iterator first, iterator last);  
void clear() ;
```

Métodos find, count,..

```
iterator find(const key_type& x) ;  
size_type count(const key_type& x) ;  
iterator lower_bound(const key_type& x) ;  
iterator upper_bound(const key_type& x) ;  
pair<iterator, iterator> equal_range(const key_type& x) ;
```


Operadores de Set

==, <,

Veja a seguir um exemplo de uso de set.

Listing 35.1: Usando set.

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

int main ()
{
    //Definição de um novo tipo usando um typedef
    //ou seja, digitar container_set
    //é o mesmo que digitar
    //std::set< double, std::less< double > >
    typedef std::set < double, std::less < double >>container_set;

    //Cria um array de C, com 4 elementos
    const int const_dimensao = 4;
    double array[const_dimensao] = { 45.12, 564.34, 347.78, 148.64 };

    //Cria um container do tipo set, para double
    container_set container (array, array + const_dimensao);;

    //Cria um iterador do tipo ostream.
    //O mesmo é usado para enviar os objetos do container para tela
    std::ostream_iterator < double >output (cout, " ");

    //Copia os elementos do container para a tela
    cout << "Conteudo do Container Set: ";
    copy (container.begin (), container.end (), output);

    //Cria um pair, um par de dados
    std::pair < container_set::const_iterator, bool > p;

    //Insere elemento no container.
    //insert retorna um par, onde o primeiro elemento é o objeto inserido e
    //o segundo um flag que indica se a inclusão ocorreu
    p = container.insert (13.8);

    //Imprime na tela, se o objeto foi ou não inserido no container
    cout << '\n' << *(p.first) << (p.second ? " foi" : " não foi") << " inserido"
        ;
    cout << "\nContainer contém: ";

    //Copia os elementos do container para a tela
    copy (container.begin (), container.end (), output);

    p = container.insert (9.5);
    cout << '\n' << *(p.first) << (p.second ? " foi" : " não foi") << "
        inserido";
    cout << "\nContainer contém: ";
    copy (container.begin (), container.end (), output);
```

```
    cout << endl;
    return 0;
}

/*
Novidades:
Uso do container set.
Uso insert,
Uso de copy para saída de dados para tela.
Uso de pair (first, second).
*/
/*
Saída:
-----
[root@mercurio Cap4-STL]# ./a.out
Conteudo do Container Set: 45.12 148.64 347.78 564.34
13.8 foi inserido
Container contains: 13.8 45.12 148.64 347.78 564.34
9.5 foi inserido
Container contém: 9.5 13.8 45.12 148.64 347.78 564.34
*/
```

Capítulo 36

class <multiset>

Um container multiset trabalha com um conjunto de chaves que podem ser repetidas. O container multiset armazena e recupera o valor da chave rapidamente. Multiset suporta iterator bidirecional.

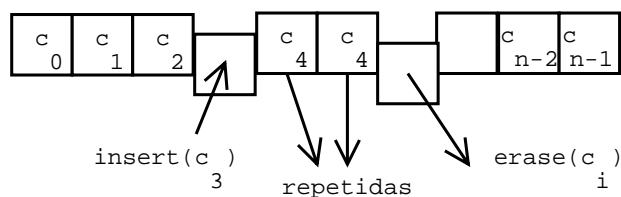
- A diferença para set é que insert retorna um iterator e não um par.
- As chaves estão sempre ordenadas, em ordem crescente.
- É um container associativo.
- É o equivalente de multimap para set.
- Para usar multiset inclua o header:

```
#include <multiset>
```

A Figura 36.1 mostra os métodos disponibilizados para multiset.

Figura 36.1: Métodos disponibilizados para multiset.

Multiset armazena um conjunto de chaves que podem ser repetidas:



Contrutores de multiset

```
multiset();  
explicit multiset(const Compare& comp);  
multiset(InputIterator first, InputIterator last);
```

```
multiset(InputIterator first, InputIterator last, const Compare& comp);  
multiset(const value_type* first, const value_type* last);  
multiset(const value_type* first, const value_type* last, const Compare& comp);  
multiset(const_iterator first, const_iterator last);  
multiset(const_iterator first, const_iterator last, const Compare& comp);  
multiset(const multiset<Key, Compare, Alloc>& x) ;  
multiset<Key, Compare, Alloc>&
```

Operadores de multiset

```
operator=(const multiset<Key, Compare, Alloc>& x) ;
```

Capítulo 37

class <map>

Neste capítulo vamos descrever o container associativo map. Entretanto, antes de descrever o container map, vamos, rapidamente falar de pair.

37.1 pair

Um pair é um objeto composto de dois outros objetos.

- Os tipos dos objetos é tipo1 e tipo2.
- Um pair é usado em alguns métodos de map e multimap.
- Um pair pode ser usado quando se deseja retornar um par de objetos de um método.

Para criar um pair faça:

```
pair <tipo1, tipo2> obj_par (valor_tipo1, valor_tipo2);
```

Para usar um pair faça:

```
cout << "primeiro objeto = " << obj_par->first();  
cout << "Segundo objeto = " << obj_par->second();
```

37.2 map

Um container map trabalha com um conjunto de chaves e de objetos associados a estas chaves, ou seja, trabalha com pares onde a ordenação e tomada de decisões é baseada nas chaves. O container map é muito útil, como veremos através do exemplo. Num map as chaves não podem ser repetidas.

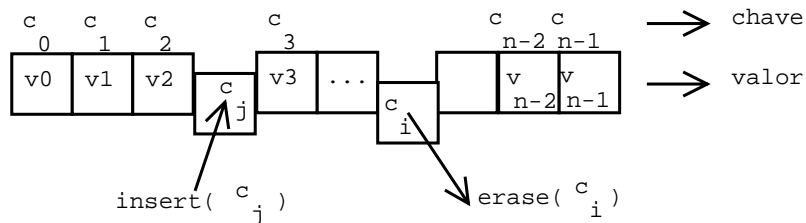
- É um container associativo.
- Para usar map inclua o header

```
#include <map>
```

A Figura 37.1 mostra os métodos disponibilizados para map.

Figura 37.1: Métodos disponibilizados para map.

Map armazena um conjunto de valores que são acessados usando-se como índice uma chave:



typedefs

```
typedef Key key_type;
typedef T data_type;
typedef T mapped_type;
typedef pair<const Key, T> value_type;
typedef Compare key_compare;
```

Construtores

```
map();
map(InputIterator first, InputIterator last);
map(InputIterator first, InputIterator last, const Compare& comp);
map(const value_type* first, const value_type* last);
map(const value_type* first, const value_type* last, const Compare& comp);
map(const_iterator first, const_iterator last);
map(const_iterator first, const_iterator last, const Compare& comp);
map(const map<Key, T, Compare, Alloc>& x);
map<Key, T, Compare, Alloc>& operator=(const map<Key, T, Compare, Alloc>& x);
```

Métodos de acesso

```
key_compare key_comp();
value_compare value_comp();
```

Métodos insert e erase

Inserir um par.

```
pair<iterator, bool> insert(const value_type& x);
iterator insert(iterator position, const value_type& x);
```

```

template <class InputIterator>
void insert(InputIterator first, InputIterator last) ;
void insert(const value_type* first, const value_type* last)
void insert(const_iterator first, const_iterator last) ;
void insert(InputIterator first, InputIterator last) ;
void insert(const value_type* first, const value_type* last);

```

Apaga elemento.

```
void erase(iterator position) ;
```

Apaga elementos com chave x.

```

size_type erase(const key_type& x);
void erase(iterator first, iterator last);
void clear() ;

```

Localiza objeto com a chave x, se não achar retorna end().

```

iterator find(const key_type& x) ;
const_iterator find(const key_type& x) ;

```

Conta o número de objetos com a chave x

```
size_type count(const key_type& x) ;
```

Último elemento menor que x

```

iterator lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) ;

```

Último elemento maior que x

```

iterator upper_bound(const key_type& x) ;
const_iterator upper_bound(const key_type& x) ;

```

Retorna um par first/second

```

pair<iterator,iterator> equal_range(const key_type& x) ;
pair<const_iterator,const_iterator> equal_range(const key_type& x) ;

```

Operadores para map

```
==, <
```

37.3 Sentenças para map

- Use um map para implementar um dicionário.
- Um map fornece iteradores bidirecionais.
- Em um map os dados são armazenados de forma ordenada, pelo operador menor que (<).

- `begin()` aponta para primeira chave ordenada (de forma crescente), `end()` aponta para última chave.
- O método `erase` retorna o número de objetos deletados.
- O retorno de `insert` é um `pair<iterator, bool>`.
 - `first` é o primeiro elemento (a chave).
 - `second` é o segundo elemento (o valor).
- Se a chave não for localizada, `map` insere a chave da pesquisa no `map`. Associando o valor a um valor default (zerado).
- Se você quer achar um par (chave, valor) mas não tem certeza de que a chave está no `map` use um `find`. Pois, como dito na sentença anterior, se a chave não for localizada ela será inserida.
- A pesquisa no `map` usando a chave tem um custo de $\sim \log(\text{tamanho_do_map})$.
- Um construtor de `map` pode receber um método de comparação de forma que você pode construir dois `maps` e usar o mesmo método de comparação em ambos.
- Na sequência AAAB, a chamada a `lower_bound()` retorna um iterator para o primeiro A, e `upper_bound()` para o primeiro elemento depois do último A, ou seja, para B.
- O método `equal_range` retorna um par onde `first` é o retorno de `lower_bound()` e `second` o retorno de `upper_bound()`.

Exemplo:

```
map<TChave, TValor>::const_iterator it;
for(it=obj.begin(); it != obj.end(); it++)
  cout << "chave=" << it->first << " valor=" << it->second;
```

Apresenta-se a seguir um exemplo de `map`. Observe na saída que os objetos do container são listados em ordem alfabética. Isto ocorre porque `map` é sempre ordenado, a ordenação é feita pela chave.

Listing 37.1: Usando `map`.

```
//-----
  TTelefone.h
#include <fstream>
#include <iomanip>
#include <string>
#include <map>

using namespace std;

//Tipo telefone
//Armazena o prefixo e o número
class TTelefone
{
```



```

private:
    int prefixo;
    int numero;
public:
    TTelefone(){prefixo=numero=0;};
    //cast
    //bool(){ return numero!=0? true:false;};

    //Sobrecarga operador para acesso a tela/teclado
    friend istream& operator>>(istream& is,TTelefone& t);
    friend ostream& operator<<(ostream& os,const TTelefone& t) ;
    //disco
    friend ofstream& operator<<(ofstream& os,const TTelefone& t) ;
};

//-----
    TTelefone.cpp
//Funcoes friend
istream& operator>>(istream& is,TTelefone& t)
{
    is >> t.prefixo;
    is >> t.numero; cin.get();
    return is;
}
ostream& operator<<(ostream& os,const TTelefone& t)
{
    os << "("<< t.prefixo<<")-"<< t.numero<<endl;
    return os;
}

//Sobrecarga operador << para saída em disco da classe TTelefone
ofstream& operator<<(ofstream& os,const TTelefone& t)
{
    os << t.prefixo<<','<< t.numero<<endl;
    return os;
}

//-----main.
    cpp
int main()
{
    //Usa um typedef, um apelido para std::map usando string e TTelefone
    typedef std::map< string, TTelefone > container_map;

    //Cria um objeto container com nome listatelefonos
    container_map listatelefonos;

    string linha ("
        -----\n");

    //Cria objeto telefone
    TTelefone telefone;

    int resp=0;

```

```

do
{
string nome;
cout<<"Entre com o nome da pessoa/empresa:"<<endl;
getline(cin,nome);

cout<<"Entre com o telefone (prefixo numero) (ctrl+d para encerrar entrada):
"<<endl;
cin>>telefone;

//Observe a inserção da chave (o nome) e do valor (o telefone).
if(cin.good())
listatelefonos.insert( container_map::value_type( nome, telefone ) );
}
while (cin.good() );
cin.clear();

cout << linha<< "Conteúdo do container:\n" << linha<< endl;
cout << "chave_ _ _ _ _ valor"<<endl;

//determinação do maior campo
container_map::const_iterator iter;
int campo=0;
for ( iter = listatelefonos.begin(); iter != listatelefonos.end(); ++iter )
{
if( iter->first.size() > campo )
campo = iter->first.size();
}

//saída para tela
cout.setf(ios::left);
for ( iter = listatelefonos.begin(); iter != listatelefonos.end(); ++iter )
cout << iter->first << " " << iter->second ;

//saída para disco
ofstream fout("Lista_telefonos_map.dat");
if(fout)
{
for ( iter = listatelefonos.begin(); iter != listatelefonos.end(); ++
iter )
fout << setw(campo)<<iter->first << iter->second ;
fout.close();
}

cout << endl;
return 0;
}

/*
Novidade:
Uso do container map.
*/
/*
Saída:
-----

```

```
[andre@mercurio Cap4-STL]$ ./a.out
Entre com o nome da pessoa/empresa:
Volvo
Entre com o telefone (prefixo numero) (ctrl+d para encerrar entrada):
048 3332516
Entre com o nome da pessoa/empresa:
Scania
Entre com o telefone (prefixo numero) (ctrl+d para encerrar entrada):
051 5148263
Entre com o nome da pessoa/empresa:
Fiat
Entre com o telefone (prefixo numero) (ctrl+d para encerrar entrada):
065 2154812
Entre com o nome da pessoa/empresa:
Entre com o telefone (prefixo numero) (ctrl+d para encerrar entrada):

-----
Conteúdo do container:
-----

chave      valor
Fiat (65)-2154812
Scania (51)-5148263
Volvo (48)-3332516
*/
```


Capítulo 38

class <multimap>

O container `multimap` trabalha com um conjunto de chaves e de objetos associados a estas chaves, ou seja, trabalha com pares. A ordenação e tomada de decisões é baseada nas chaves. Num `multimap` as chaves podem ser repetidas.

- Não tem operador de subscripto.
- É um container associativo.
- Em `multimap`, `insert` retorna um iterator e não um pair.
- Para acessar os valores que tem a mesma chave, usa-se `equal_range()`, `lower_bound()` e `upper_bound()`.
 - `lower_bound(x)`
Menor valor menor que x.
 - `upper_bound(x)`
Maior valor maior que x.
 - `equal_range(x)`
Retorna um par dado por `first = lower_bound()`, `second = upper_bound()`.
- Para usar um `multimap` inclua o arquivo de cabeçalho `<map>`.
 - Exemplo:
 - `#include <map>`

Capítulo 39

Algoritmos Genéricos

39.1 Introdução aos algoritmos genéricos

A STL fornece uma biblioteca de classes para manipulação de container's e provê um rico conjunto de algoritmos para pesquisa, ordenação, mistura, troca, e transformações em um container.

- Cada algoritmo pode ser aplicado a um conjunto específico de container's.
- De uma maneira geral, as funções genéricas usam iteradores para acessar e manipular os container's.
- Os algoritmos genéricos foram construídos de forma a necessitar de um número reduzido de serviços dos iteradores.
- Alguns algoritmos genéricos exigem que a classe container já tenha sido ordenada.

39.2 Classificação dos algoritmos genéricos

Apresenta-se a seguir uma classificação dos algoritmos genéricos quanto a mudança do container, quanto ao tipo das operações e quanto ao iterador necessário.

39.2.1 Classificação quanto a modificação do container

Funções que não mudam o container

accumulate, find, max, adjacent_find, find_if, max_element, binary_search, find_first_of, min, count, for_each, min_element, count_if, includes, mismatch, equal, lexicographical_compare, nth_element, equal_range, lower_bound, mismatch, search

Funções que mudam o container

copy, remove_if, copy_backward, replace, fill, replace_copy, fill_n, replace_copy_if, generate, replace_if, generate_n, reverse, inplace_merge, reverse_copy, iter_swap, rotate, rotate_copy, merge, set_difference, nth_element, set_symmetric_difference,

next_permutation, set_intersection, partial_sort, set_union, partial_sort_copy, sort, partition, sort_heap, prev_permutation, stable_partition, push_heap, stable_sort, pop_heap, swap, random_shuffle, swap_ranges, remove, transform, remove_copy, unique, remove_copy_if, unique_copy

39.2.2 Classificação quando as operações realizadas

Operações de inicialização: fill, generate, fill_n, generate_n

Operações de busca: adjacent_find, find_if, count, find_first_of, count_if, search, find

Operações de busca binária: binary_search, lower_bound, equal_range, upper_bound

Operações de comparação: equal, mismatch, lexicographical_compare, Copy operations, copy, copy_backward

Operações de transformação: partition, reverse, random_shuffle, reverse_copy, replace, rotate, replace_copy, rotate_copy, replace_copy_if, stable_partition, replace_if, transform

Operações de troca(swap): swap, swap_ranges

Operações de scanning: accumulate, for_each

Operações de remoção: remove, remove_if, remove_copy, unique, remove_copy_if, unique_copy

Operações de ordenação: nth_element, sort, partial_sort, stable_sort, partial_sort_copy

Operações de mistura: inplace_merge, merge

Operações de set (conjuntos): includes, set_symmetric_difference, set_difference, set_union, set_intersection

Operações de pilha (Heap operations): make_heap, push_heap, pop_heap, sort_heap

Mínimo e máximo: max, min, max_element, min_element

Permutação genérica: next_permutation, prev_permutation

39.2.3 Classificação quanto a categoria dos iteradores

Algoritmos que não usam iterador: max, min, swap

Requer somente input_iterator: accumulate, find, mismatch, count, find_if, count_if, includes, equal, inner_product, for_each, lexicographical_compare

Requer somente output_iterator: fill_n, generate_n

Lê de um input_iterator e escreve para um output_iterator: adjacent_difference, replace_copy, transform, copy, replace_copy_if, unique_copy, merge, set_difference, partial_sum, set_intersection, remove_copy, set_symmetric_difference, remove_copy_if, set_union

Requer um forward iterator: `adjacent_find`, `lower_bound`, `rotate`, `binary_search`, `max_element`, `search`, `equal_range`, `min_element`, `swap_ranges`, `fill`, `remove`, `unique`, `find_first_of`, `remove_if`, `upper_bound`, `generate`, `replace`, `iter_swap`, `replace_if`

Lê de um forward_iterator e escreve para um output_iterator: `rotate_copy`

Requer um bidirectional_iterator: `copy_backward`, `partition`, `inplace_merge`, `prev_permutation`, `next_permutation`, `reverse`, `stable_permutation`

Lê de um iterator bidirecional e escreve em um output_iterator: `reverse_copy`

Requer um iterator randomico: `make_heap`, `pop_heap`, `sort`, `nth_element`, `push_heap`, `sort_heap`, `partial_sort`, `random_shuffle`, `stable_sort`

Lê de um input_iterator e escreve para um random_access_iterator: `partial_sort_copy`

39.3 Funções genéricas

Apresenta-se a seguir uma descrição mais detalhada de cada algoritmo genérico.

39.3.1 Preenchimento

Preencher de `begin` a `end` com valor.

```
fill(v.begin(),v.end(), valor);
```

Preencher de `begin` até `n` com valor.

```
fill_n(v.begin(),n, valor);
```

De `begin` a `end` executa a função (void funcao()).

```
generate(v.begin(),v.end(), funcao);
```

De `begin` a `n` executa a função.

```
generate_n(v.begin(),n, funcao);
```

Copia de `first` a `last` para `out`. Use `copy` para gerar uma saída do container ou para gerar um novo container.

```
copy(first, last, out);
```

Copia os elementos de `v` para `r`, `v[0]` é colocado no fim de `r`, e assim sucessivamente (terminando no inicio de `r`).

```
copy_backward(v.begin(),v.end(),r.end())
```

Copia de `first` a `last` para `out`

```
copy(first, last, out);
```

39.3.2 Comparação

Compara a igualdade de cada elemento do container `v` e `c`. Retorna `true` se for tudo igual.

```
bool r = equal(v.begin(),v.end(),c.begin());
```

`pair < vector<int>::iterator,vector<int>::iterator >` local;

Mismatch faz.....

```
local = mismatch(v.begin(),v.end(),c.begin());
```

Retorna posição onde os containers `v` e `c` são diferentes.

```
lexicographical_compare(v,v+size, c,c+size);
```

Retorna `true` se `v` é maior que `c` (em caracteres)(?).

Retorna `a` se `a>b`, ou `b` se `b>a`. Retorna o maior valor.

```
max(a,b);
```

Retorna `a` se `a<b`, ou `b` se `b<a`. Retorna o menor valor.

```
min(a,b);
```

39.3.3 Remoção

Remove de `begin` a `end` o valor.

```
remove(v.begin(),v.end(), valor);
```

Remove de `begin` a `end` se a função retornar `true`.

```
remove_if(v.begin(),v.end(), funcao);
```

Remove de `begin` a `end` o valor e copia para `c`.

Se o container `c` for pequeno vai haver estouro de pilha.

```
remove_copy(v.begin(),v.end(),c.begin(), valor);
```

Remove de `begin` a `end` se a função retornar `true` e copia para `c`.

```
remove_copy_if(v.begin(),v.end(),c.begin(), funcao);
```

39.3.4 Trocas

Troca de `begin` a `end` valor por `novoValor`.

```
replace(v.begin(),v.end(), valor, novoValor);
```

Se a função retornar `true`, troca pelo `novoValor`.

```
replace_if(v.begin(),v.end(), funcao, novoValor);
```

Troca de `begin` a `end` valor por `novoValor` e copia para `c`.

```
replace_copy(v.begin(),v.end(),c.begin(), valor,novoValor);
```

Se a função retornar true, troca por novoValor e copia para c.

```
replace_copy_if(v.begin(),v.end(),c, funcao,novoValor);
```

Troca v[0] por v[1].

```
swap(v[0],v[1]);
```

Troca os objetos apontados por it1 e it2.

```
iter_swap(it1 ,it2 );
```

Elimina o intervalo de v a v+3 e copia valores a partir de v+4.

Troca os objetos no intervalo especificado.

```
swap_range(v,v+3,v+4);
```

39.3.5 Misturar/Mesclar/Inverter

Pega v1 e v2 ordenados e cria vetor v3 com todos os elementos de v1 e v2, retornando v3.

```
merge(v1.begin(),v1.end(),v2.begin(),v2.end(),resultado.begin())
```

Mistura dois conjuntos de dados do mesmo container, vai misturar os valores de begin a begin+n com os valores a partir de begin+n (sem ultrapassar v.end()).

```
inplace_merge(v.begin(),v.begin()+n, v.end())
```

Muda a ordem, o primeiro passa a ser o último.

```
reverse(v.begin(),v.end());
```

Inverte os elementos de v e copia para r.

```
reverse_copy(v.begin(),v.end(),back_inserter(r));
```

Compara os container's ordenados a e b, se qualquer elemento de b estiver presente em a retorna true.

```
if (includes(a,a+size,b,b+zise))
```

39.3.6 Pesquisa, ordenação

Procurar de begin a end o valor.

```
find(v.begin(),v.end(), valor);
```

Procurar de begin a end o elemento que satisfaz a função.

```
find_if(v.begin(),v.end(), funcao);
```

Retorna it para elemento de P que existe em C.

```
find_first_of(p.begin(),p.end(),c.begin(),c.end());
```

Procura por um par de valores iguais e adjacentes, retorna it para o primeiro elemento.

adjacent_find(first,last);

Ordena o vetor v.

sort(v.begin(),v.end());

PS; sort não é disponível para list, use o sort do próprio container list.

Ordena do início até o meio.

partial_sort(inicio,meio,fim);

Retorna true se o valor esta presente no container

if(binary_search(v.begin(),v.end(),valor))

Retorna um iterador para elemento menor que valor.

vector<int>::iterator lower;

lower = lower_bound(v.begin(),v.end(),valor);

Para manter o container ordenado inserir valor logo após esta posição.

Se tiver 3,4, 6,12,34,34,34,50 e valor=34 retorna iterador para o primeiro 34

Retorna um iterador para elemento maior que valor.

vector<int>::iterator uper;

uper = uper_bound(v.begin(),v.end(),valor);

Se tiver 3,4, 6,12,34,34,50 e valor=34 retorna iterador para o último 34.

pair<vector<int>::iterator, vector<int>::iterator,> eq;

Retorna um pair para aplicação de first=lower_bound e second=uper_bound.

eq = equal_range(v.begin(),v.end(),valor);

Rotaciona ciclicamente.

rotate(inicio,fim,out);

rotate_copy();

Procuram uma sequência de first-last que exista em first2-last2. Ou seja, a sequência do container 1 existe no container 2?. Retorna iterador para primeiro objeto no container1.

search(first,last,first2,last2);

search(first,last, predicado);

search_n: procura sequência com n combinações.

O find procura por valor no intervalo especificado.

find(first,last,valor);

find_end: realiza o mesmo que search, mas na direção inversa.

O find_if procura no intervalo first-last o objeto que satisfaça o predicado.

find_if(first,last, predicado);

adjacent_find procura por um par de objetos iguais

```
adjacent_find(first,last);
```

Mistura randomicamente os elementos do container.

```
void random_shuffle(v.begin(),v.end());
```

Exemplo:

```
string v[]={‘a’,‘b’,‘c’};
reverse(v, v + 3 ); // fica: c,a,b
```

39.3.7 Classificação

Pega v1 ordenado, e v2 ordenado e cria um vetor v3 com todos os elementos de v1 e v2, retornando v3, e depois copiando v3 para r usando push_back.

```
merge(v1.begin(),v1.end(),v2.begin(),v2.end(),back_inserter(r))
```

```
vector<int>"iterador localizacaoFinal;
```

Obtém uma cópia de v, sem elementos repetidos e copia para r.

```
unique_copy(v.begin(),v.end(),back_inserter(r));
```

Elimina os elementos duplicados, movendo os mesmos para o fim do container.

Retorna iterador para último elemento não duplicado.

PS: antes de chamar unique, chame sort.

```
localizacaoFinal = unique(v.begin(),v.end());
```

Exemplo:

```
//Para realmente eliminar os elementos duplicados.
//use sort para ordenar o container
sort(c.begin(),c.end());
//use unique para mover para trás os elementos duplicados
iterator p = unique(c.begin(),c.end());
//use container.erase(p , v.end()); para deletar elementos duplicados
c.erase(p,v.end());
```

39.3.8 Matemáticos

Armazena de begin a end números randomicos.

```
random_shuffle(v.begin(),v.end());
```

Determina o número de elementos igual a valor.

```
int total = count(v.begin(),v.end(),valor);
```

Determina o número de elementos que obedecem a função.

```
int total = count_if(v.begin(),v.end(),funcao);
```

Retorna o maior elemento.

```
max_element(v.begin(),v.end());
```

Retorna o menor elemento.

```
min_element(v.begin(),v.end());
```

Retorna a soma de todos os elementos.

```
int somatório = accumulate(v.begin(),v.end(),0);
```

De begin e end executa a função, usada para aplicar uma dada função a todos os elementos do container, exceto o v.end().

```
for_each(v.begin(),v.end(), funcao);
```

De v.begin a v.end executa a função e armazena o resultado em c.begin.

Observe que poderia-se usar:

```
transform(v.begin(),v.end(), v.begin(), funcao);
```

```
transform(v.begin(),v.end(), c.begin(), funcao);
```

39.3.9 Operações matemáticas com conjuntos

Todos os valores do vetor a que não estiverem no vetor b serão copiados para o vetor diferenca.

```
int diferenca[size];
```

```
int*ptr = set_difference (a,a+n,b,b+n,diferenca);
```

Todos os valores do vetor a que estiverem no vetor b serão copiados para o vetor intersecao.

```
int intersecao[size];
```

```
int*ptr = set_intersection (a,a+n,b,b+n,intersecao);
```

Todos os valores do vetor a e b serão copiados para o vetor uniao.

```
int uniao[size];
```

```
int*ptr = set_union (a,a+n,b,b+n,uniao);
```

Determina o conjunto de valores de a que não estão em b, e os valores de b que não estão em a, e copia para o vetor symmetric_dif.

```
int sym_dif[size];
```

```
ptr = set_symmetric_difference(a,a+size,b,b+size,sym_dif);
```

39.3.10 Heapsort

Marca a pilha?.

```
make_heap(v.begin(),v.end());
```

Ordena a pilha.

```
sort_heap(v.begin(),v.end());
```

Coloca valor na pilha.

```
v.push_back(valor);
```

Coloca na pilha.

```
push_heap(v.begin(),v.end());
```

Retira elemento do topo da pilha.

```
pop_heap(v.begin(),v.end());
```

Sentenças para algoritmo genérico:

- Partições: uma partição ordena o container de acordo com um predicado. Vão para o início do container os objetos que satisfazem o predicado.
`partition(inicio, fim, predicado);`
- `min` e `max` retornam o menor e maior valor do container.
- `next_permutation` e `prev_permutation` permutam os elementos do container.
- De uma olhada na internet e procure por `stl`. você vai encontrar muitos sites com exemplos interessantes.

39.3.11 Exemplos

Veja a seguir exemplos de uso de algoritmos genéricos.

Listing 39.1: Usando algoritmos genéricos.

```
//Classes de entrada e saída
#include <iostream>

//Classe de listas
#include <list>

//Algoritmo genérico
#include <algorithm>

//Uso de namespace
using namespace std;

//Definição da função main
int
main ()
{
    //Cria um iterador para ostream
    ostream_iterator < float > output (cout, "\n");

    //Criação de duas listas para inteiros
    std::list < float > container_list, container_list2;
```

```

//Inclue valores na lista
container_list.push_front (312.1f);
container_list.push_back (313.4f);
container_list.push_back (316.7f);
container_list.push_front (312.1f);

//Mostra lista
cout << "\nConteúdo do container:" << endl;
copy (container_list.begin (), container_list.end (), output);

//Ordena lista
container_list.sort ();
cout << "\nConteúdo do container após sort:" << endl;
copy (container_list.begin (), container_list.end (), output);

//Função splice (Adiciona ao final de container_list
//os valores de container_list2)
container_list.splice (container_list.end (), container_list2);
cout << "\nConteúdo do container após splice";
cout <<(container_list.end (), container_list2);:" <<endl;
copy (container_list.begin (), container_list.end (), output);

//Ordena a lista
container_list.sort ();
cout << "\nConteúdo do container após sort:" <<endl;
copy (container_list.begin (), container_list.end (), output);

//Adiciona elementos a lista2
container_list2.push_front (22.0);
container_list2.push_front (2222.0);
cout << "\nConteúdo do container 2:\n ";
copy (container_list2.begin (), container_list2.end (), output);

//Mistura as duas listas, colocando tudo em container_list
//e eliminando tudo de container_list2
container_list.merge (container_list2);
cout << "\nConteúdo do container após container_list.merge (container_list2):\n";
copy (container_list.begin (), container_list.end (), output);
cout << "\nConteúdo do container 2: " <<endl;
copy (container_list2.begin (), container_list2.end (), output);

//Elimina valores duplicados
container_list.unique ();
cout << "\nContainer depois de unique : \n";
copy (container_list.begin (), container_list.end (), output);

//Chama funções pop_front e pop_back
container_list.pop_front (); //elimina primeiro elemento da lista
container_list.pop_back (); //elimina ultimo elemento da lista
cout << "\nContainer depois de pop_front e pop_back: \n";
copy (container_list.begin (), container_list.end (), output);

//Troca tudo entre as duas listas

```



```

    container_list.swap(container_list2);
    cout << "\nContainer depois de swap entre as duas listas:\n";
    copy(container_list.begin(), container_list.end(), output);

    cout << "\nContainer_list2 contém:\n";
    copy(container_list2.begin(), container_list2.end(), output);

    //Atribue valores de container_list2 em container_list
    container_list.assign(container_list2.begin(), container_list2.end());
    cout << "\nContainer depois de \ncontainer_list.assign ";
    cout << (container_list2.begin(), container_list2.end());\n ";
    copy(container_list.begin(), container_list.end(), output);

    //Mistura novamente
    container_list.merge(container_list2);
    cout << "\nContainer depois de novo merge:\n";
    copy(container_list.begin(), container_list.end(), output);

    //Remove elemento?
    container_list.remove(2);
    cout << "\nContainer após remove( 2 ) container_list contém:\n";
    copy(container_list.begin(), container_list.end(), output);
    cout << endl;
    cin.get();
    return 0;
}

/*
Novidades:
Uso de copy, sort, splice, merge, pop_front, pop_back,
unique, swap, merge, remove
*/
/*
Dica:
Particularmente não gosto do uso do ostream_iterator,
prefiro sobre carregar os operadores de extração << e inserção >>.
Afim de contas, o código
cout << list << endl;

é bem mais limpo e claro que
cout << copy(list.begin(), list.end(), output);
*/
/*
Saída:
-----
[andre@mercurio_Cap4-STL]$ ./a.out

Conteúdo do container:
312.1 312.1 313.4 316.7
Conteúdo do container após sort:
312.1 312.1 313.4 316.7
Conteúdo do container após splice(container_list.end(), container_list2);:
312.1 312.1 313.4 316.7
Conteúdo do container após sort:
312.1 312.1 313.4 316.7

```

```

Conteúdo do container 2:
 2222 22
Conteúdo do container após container_list.merge (container_list2);:
 312.1 312.1 313.4 316.7 2222 22
Conteúdo do container 2:

Container depois de unique :
312.1 313.4 316.7 2222 22
Container depois de pop_front e pop_back:
313.4 316.7 2222
Container depois de swap entre as duas listas:

Container_list2 contém:
313.4 316.7 2222
Container depois de
container_list.assign (container_list2.begin (), container_list2.end ());
 313.4 316.7 2222
Container depois de novo merge:
313.4 313.4 316.7 316.7 2222 2222
Container após remove( 2 ) container_list contem:
*/

```

Listing 39.2: Usando vector com algoritmos genéricos

```

#include <fstream>
#include <iomanip>
#include <string>
#include <vector>           //Classe de vetores
#include <algorithm>       //Classe para algoritmos genéricos

using namespace std;      //Define estar usando espaço de nomes std

//Funções globais
ostream & operator<< (ostream & os, const vector < int >&v);
ofstream & operator<< (ofstream & os, const vector < int >&v);

//Declaração de função predicado
bool maiorQue5 (int value)
{
    return value > 5;
};

//Definição da função main
int
main ()
{
    string linha =
        "-----\n";
    //Cria vector, do tipo int, com nome v
    vector < int >v;

    int data;
    do
    {
        cout << "\nEntre com o dado (" << setw (3) << v.size () << "):";
        cin >> data;
    }
}

```

```

        cin.get ();
        if(cin.good ())
            v.push_back (data);
    }
while (cin.good ());
cin.get ();
cin.clear ();                //corrige o cin

{
    ofstream fout ("vector.dat");
    if (!fout)
        return 0;
    fout << v << endl;
    fout.close ();
}

cout << "\n";
cout << linha << v << endl;

//Chama função erase do objeto vector passando v.begin
int numero;
cout << "\nEntre com o número a ser localizado:";
cin >> numero;
cin.get ();

//Ponteiro para a posição localizada
int *posicao = find (v.begin (), v.end (), numero);
cout << "\nNúmero localizado na posição:" << posicao << endl;

//Localiza primeiro elemento que satisfaz a condição dada pela função
maiorQue5
posicao = find_if (v.begin (), v.end (), maiorQue5);
cout << "\nNúmero maior que 5 localizado na posição:" << posicao << endl;

//Ordena o contêiner
sort (v.begin (), v.end ());
cout << "\nVetor após ordenação com sort(v.begin(),v.end())" << endl;
cout << linha << v << endl;

//Preenche com o valor 45
fill (v.begin (), v.end (), 45);
cout << "\nVetor após fill(v.begin(),v.end(),'45');" << endl;
cout << linha << v << endl;

//Retorna dimensão e capacidade
cout << "v.size()" << v.size () << endl;
cout << "v.capacity()" << v.capacity () << endl;

//Redimensiona o container
v.resize (20);
cout << "\nVetor após resize(20):" << endl;
cout << linha << v << endl;
cout << "v.size()" << v.size () << endl;
cout << "v.capacity()" << v.capacity () << endl;

```

```

    cout << linha << endl;
    cin.get ();
    return 0;
}

ostream & operator<< (ostream & os, const vector < int >&v)
{
    for (int i = 0; i < v.size (); i++)
    {
        os << "v[" << setw (3) << i << "]= " << setw (5) << v[i] << ' ';
    }
    return os;
}

ofstream & operator<< (ofstream & os, const vector < int >&v)
{
    for (int i = 0; i < v.size (); i++)
        os << setw (10) << v[i] << endl;
    return os;
}

/*
Novidades:
Uso de cin.clear
Uso de find, find_if e fill.
Uso de sort
Uso de size e capacity.
Uso de resize
*/

/*Saída:
-----
[andre@mercurio Cap4-STL]$ ./a.out
Entre com o dado ( 0):0
Entre com o dado ( 1):-1
Entre com o dado ( 2):-2
Entre com o dado ( 3):-3
Entre com o dado ( 4):-4
Entre com o dado ( 5):
-----
v[ 0]= 0 v[ 1]= -1 v[ 2]= -2 v[ 3]= -3 v[ 4]= -4

Entre com o número a ser localizado:-2
Número localizado na posição:0x804e680
Número maior que 5 localizado na posição:0x804e68c
Vetor após ordenação com sort(v.begin(),v.end())
-----
v[ 0]= -4 v[ 1]= -3 v[ 2]= -2 v[ 3]= -1 v[ 4]= 0

Vetor após fill( v.begin(), v.end(), '45' );
-----
v[ 0]= 45 v[ 1]= 45 v[ 2]= 45 v[ 3]= 45 v[ 4]= 45
v.size()=5
v.capacity()=8

```

Vetor após resize(20):

```
-----  
v[ 0]= 45 v[ 1]= 45 v[ 2]= 45 v[ 3]= 45 v[ 4]= 45  
v[ 5]= 0 v[ 6]= 0 v[ 7]= 0 v[ 8]= 0 v[ 9]= 0  
v[10]= 0 v[11]= 0 v[12]= 0 v[13]= 0 v[14]= 0  
v[15]= 0 v[16]= 0 v[17]= 0 v[18]= 0 v[19]= 0  
v.size()=20  
v.capacity()=20  
-----  
*/
```


Capítulo 40

Objetos Funções da STL

Algumas classes podem ter o operador `()` sobrecarregado. Desta forma, pode-se fazer:

```
Tipo NomeObjeto;  
int res = NomeObjeto(parâmetro);
```

ou seja, o objeto se comporta como sendo uma função.

Como este procedimento é muito usado, a STL, inclui classes para objetos função. Uma para funções que recebem um parâmetro e outra para funções que recebem dois parâmetros.

- Para usar o template `<functional>` inclua o header

```
# include<functional>
```

40.1 Introdução aos objetos funções da STL

Como dito, funções com um argumento são funções unárias e com dois argumentos binárias.

Alguns objetos funções fornecidos pela STL são listados a seguir.

40.1.1 Funções aritméticas

plus addition ($x + y$)

minus subtraction ($x - y$)

times multiplication ($x * y$)

divides division (x / y)

modulus remainder ($x \% y$)

negate negation ($- x$)

40.1.2 Funções de comparação

`equal_to` equality test `x == y`

`not_equal_to` inequality test `x != y`

`greater` greater comparison `x > y`

`less` less-than comparison `x < y`

`greater_equal` greater than or equal comparison `x >= y`

`less_equal` less than or equal comparison `x <= y`

40.1.3 Funções lógicas

`logical_and` logical conjunction `x && y`

`logical_or` logical disjunction `x || y`

`logical_not` logical negation `! x`

Veja a seguir um exemplo de uso de `<functional>`.

Listing 40.1: Usando `functional`.

```
//-----Includes
# include<functional>
# include<deque>
# include<vector>
# include<algorithm>

using namespace std;

//-----Classe Funcao
//Cria uma função objeto a partir de uma função unária
template<class Arg>
class TFatorial : public unary_function<Arg, Arg>
{
public:
Arg operator()(const Arg& arg)
{
    Arg a = 1;
    for(Arg i = 2; i <= arg; i++)
        a *= i;
    return a;
}
};

//-----Main
int main()
```



```

{
//Inicializa um array de C
int array[7] = {1,2,3,4,5,6,7};

//Cria um deque a partir de um array de C
deque<int> d(array, array + 7);

//Cria um vetor vazio para armazenar os fatoriais
vector<int> v((size_t)7);

//Determina o fatorial e armazena no vetor
transform(d.begin(), d.end(), v.begin(), TFatorial<int>());

//Mostra resultados
cout << "Números:" << endl << "\n";
copy(d.begin(), d.end(), ostream_iterator<int>(cout, "\n"));
cout << endl << endl;

cout << "\ne fatoriais:" << endl << "\n";
copy(v.begin(), v.end(), ostream_iterator<int>(cout, "\n"));
cout << endl << endl;

//-----
char resp;
TFatorial<int> objeto_funcao;
do
{
    cout << "Entre com um número (int):";
    int numero;
    cin >> numero; cin.get();
    cout << "Número=" << numero << "\nfatorial=" << objeto_funcao(numero) << endl;
    cout << "Continuar (s/n)?";
    cin.get(resp); cin.get();
}
while(resp == 's' || resp == 'S');
return 0;
}
/*
Novidade:
-----
Uso de classe função (#include <functional>)
*/

/*
Saída:
-----
[andre@mercurio Cap4-STL]$ ./a.out
Números:
1 2 3 4 5 6 7

e fatoriais:
1 2 6 24 120 720 5040

Entre com um número (int):13
Número = 13 fatorial = 1932053504

```

```
Continuar (s/n)?5
[andre@mercurio Cap4-STL]$ ./a.out
Números:
 1 2 3 4 5 6 7

e fatoriais:
 1 2 6 24 120 720 5040

Entre com um número (int):5
Número = 5 fatorial = 120
Continuar (s/n)?s
Entre com um número (int):6
Número = 6 fatorial = 720
Continuar (s/n)?n
*/
```

Parte V

Programação Para Linux/Unix

Capítulo 41

Introdução a Programação GNU/Linux/Unix

Este resumo contém dicas e instruções para montagem de programas usando o formato GNU. Inicialmente apresenta-se uma lista de comandos do shell e de programas úteis no ambiente Linux. Descreve-se o programa, sua utilidade, os arquivos de configuração e os parâmetros opcionais. Quando conveniente apresenta-se um exemplo.

A seguir descreve-se o uso do gcc e do make para montagem de programas pequenos. Depois, apresenta-se um roteiro para montar programas completos usando o padrão GNU.

- O texto aqui apresentado é um texto introdutório.
- Um texto intermediário, que abrange diversos aspectos da programação para Linux (ferramentas da gnu, processos, sinais, device drives, programação em ambientes gráficos) é encontrado no livro “*Linux Programming Unleashed*” de Kurt Wall *et al.* Em português existe o “*Programando com Ferramentas GNU*” (editado pela conectiva).
- Textos avançados são os manuais disponibilizados na internet, isto é, para aprender em detalhes o autoconf, baixe o manual do autoconf, o mesmo é válido para os demais tópicos apresentados nesta apostila.

Este resumo tem como base as páginas man e os manuais públicos do make (3.78), egcs (1.1.2), egcs++ (1.1.2), aucoconf (2.13), automake (1.4), libttool¹. No manual do doxygen, e em artigos da revista do linux (<http://www.revistadolinux.com.br>).

41.1 O básico do GNU/Linux/Unix

41.1.1 Comandos do shell úteis²

Lista-se a seguir alguns comandos de shell úteis para programadores. Uma descrição mais detalhada dos mesmo pode ser encontrada em apostilas/livros sobre o Linux/Unix. Você pode obter informações simplificadas sobre estes comandos usando o programa man (de manual). Use o

¹Estes manuais são encontrados, em inglês, no site da gnu (<http://www.gnu.org>).

²Uma excelente apostila de referência sobre o linux é encontrada em <http://www.cipsga.org.br>.

comando **man nomeDoPrograma** para ver um manual simples do programa. Versões mais atualizadas dos manuais destes programas são obtidos com **info nomeDoPrograma**. Você também pode obter um help resumido usando **nomeDoPrograma - -help**.

Nos exemplos apresentados a seguir, os comentários após o sinal # não devem ser digitados. Se você já é um usuário experiente do Linux pode pular esta parte.

Diretórios

```
.
. Diretório atual.
.. Diretório pai (Ex: cd ..).
~ Diretório HOME do usuário (Ex: cd ~).
```

cd Muda diretório corrente

```
cd /home/philippi #Caminho completo
cd ../../usr      #Caminho relativo
cd -              #Alterna para diretório anterior
```

pwd Mostra a path do diretório corrente.

ls Lista o conteúdo do diretório.

```
-l      #Lista detalhada.
-a      #Mostra executável e ocultos.
-b      #Número de links do arquivo.
-m      #Mostra tudo em uma linha.
-F      #Mostra \ dos diretórios.
-x      #Mostra em colunas.
```

```
ls -F | egrep /      #mostra diretórios
```

tree Lista em árvore.

```
tree -d      #Lista somente os diretórios
```

mkdir Cria diretório.

```
mkdir test
mkdir d1 d2      #Cria diretórios d1 e d2
mkdir d1/d2      #Cria d2 e o filho d1
mkdir -p d3/d31/d32 #Cria os diretórios d3 d3/d31 d3/d31/d32
```

mvdir Move ou renomeia um diretório.

rmdir Remove diretório.

```
-R      #Recursivo, elimina subdiretórios (usar com cuidado).

rmdir -p d3/d31/d32 #Remove todos os diretórios
rm -R diretorio     #Remove o diretório e seus sub-diretórios
```

Arquivos**cp a1 a2** Cópia arquivos e diretórios.

```

-b          Cria backup de a2.
-i          Cópia iterativa.
-r          Cópia recursiva.
-P          Cópia arquivo e estrutura de diretório.
-p          Preserva as permissões e horas.
-v          Modo verbose.
-b          Cria backup.

```

```
cp a1 a2
```

mv Move arquivos (renomeia arquivos).

```

-b          Cria backup.
-v          Modo verbose.
-i          Iterativa.

mv a1 a2          #Renomeia arq a1 para a2
mv d1 d2          #Renomeia dir d1 para d2
mv -b a1 a2      #Renomeia com backup

```

rm Remove arquivos (retira links).

```

-d          Remove diretório.
-i          Remove iterativamente.
-r          Remove diretórios recursivamente.
-f          Desconsidera confirmação.

#Só execute o comando abaixo em um subdiretório sem importância
rm -f -r *          #Remove tudo (*) sem pedir confirmação

```

ln Linka arquivos e diretórios (um arquivo com link só é deletado se for o último link).

```

ln -f chap1 intro
ln origem link          #Cria link permanente
ln -s origem link      #Cria link simbólico

```

find O find é usado para pesquisar arquivos em seu HD.

```

find          path expressão
-name        Informa o nome do arquivo.

```

`-print` *Mostra a path.*
`-type` *Informa o tipo.*
`-atime` *Informa dados de data.*
`-size` *Informa tamanho(+ ou -).*
`-o` *Aceita repetição de parâmetro (Other).*
`-xdev` *Desconsidera arquivos NFS.*
`-exec [comando [opcoes]]` *Permite executar comando.*

²Exemplos:

```
#Para achar arquivos core:
find / -name core
#Para localizar arquivos do usuário:
find PATH -USER nomeUsuário
#Para localizar arquivos *.cpp:
find PATH -name *.cpp
#Para localizar e remover arquivos .o:
find PATH -name *.o | xargs rm
#Localizando tudo com a extensão *.o e *.a
find -name *.o -o -name *.a
#PS: exec só funciona com alguns programas, melhor usar xargs.
find -type f -atime +100 -print
find . -size +1000k
find ~/ -size -100k
find [a-c]????
find file[12]
```

head n Mostrar as primeiras n linhas de um arquivo.

```
head -5 nome.txt
```

tail n Exibe arquivo a partir de uma linha.

```
tail -20 notes
```

cat arq1 Mostra conteúdo do arquivo arq1.

```
cat f1 #Mostra arquivo f1
#Cria novo arquivo:
cat > texto.txt
...digita o texto...
contrl+d #Finaliza arquivo
cat a1 a2 > a3 #Concatena a1 e a2 e armazena em a3
cat a >> b #Acrescenta ao final do arquivo b o arquivo a
```

cat a1 a2 Mostra arquivos a1 e depois a2

cat -n a2 Mostra conteúdo de a2 com numeração

```
ls -la | cat -n
```

less arq Mostra conteúdo do arquivo (+completo)

```
#/str para localizar a string str no texto visualizado  
less arq #q para sair
```

file arq Informa o tipo de arquivo.

```
file *  
file * | egrep directory
```

tr Converte cadeias de caracteres em arquivos.

```
ls | tr a-z A-Z #de minúsculas para maiúsculas
```

xargs Facilita passagem de parâmetros para outro comando.

```
xargs [opções][comando [opções]]  
  
#Procura e deleta arquivos *.cpp  
xargs grep \l foo find /usr/src \name "*.cpp"  
find /tmp -name "*.cpp" | xargs rm
```

nl Número de linhas do arquivo.

wc Número de linhas, de palavras e de bytes do arquivo.

Pesquisa dentro de arquivos

grep O grep é usado para pesquisar o que tem dentro de um arquivo.

```
-n      Mostra número da linha.  
-F      O grep funciona como o fgrep.  
-c      Retorna número de coincidências.  
-i      Desconsidera maiúsculas/minusculas.  
-s      Desconsidera mensagens de erro.  
-v      Modo verbose.  
-A n    Lista também as n linhas posteriores.  
-B n    Lista também as n linhas anteriores.  
-r      Ordem inversa.  
-f      Usa arquivo auxiliar.
```

```
man grep                #Mostra detalhes do grep
greet -v buble sort.c
ls -l |greet "julia"    #Lista diretório e pesquisa pelo arquivo julia
grep ^[0-9] guide.txt
grep "(b)" guide.txt
grep arqAux guide.txt  # Pesquisa em guide.txt usando arqAux
```

sort Ordena arquivos.

```
-c      Verifica arquivo.
-o      Especifica nome arquivo saída.
-d      Ordem dicionário.
-f      Despresa diferença maiúscula/minuscula.
-t      Atualiza data e hora.
-s      Modo silencioso.
```

```
sort -r arquivo
ls | sort -r           #Ordem invertida
#Ordena a listagem de diretório a partir da 4 coluna,considerando número
ls -l |egrep rwx | sort +4n
```

Compactação e backup

zip Compatível com pzip/pkzip do DOS.

unzip Unzipa arquivos zip.

```
zip -r nome.zip nomeDiretório
unzip nome.zip
```

gzip / **gunzip** Compacta/Descompacta arquivos com a extensão: gz,.Z,-gz,.z,-z

```
-c      Mostra arquivo na tela.
-d      Descomprime o arquivo.
-S      Extensão do arquivo.
-f      Força compressão.
-l      Lista arquivos.
-r      Mostra diretórios recursivamente.
-t      Testa integridade do arquivo.
-v      Modo verbose.
-1      Mais veloz e menos compactado.
-9      Mais lento e mais compactado.
```

```
#Para compactar todo um diretório
tar -cvzf nomeDiretorio.tar.gz nomeDiretorio
#Para descompactar
tar -xvzf nomeDiretorio.tar.gz
```

bzip2 Compactador mais eficiente que o gzip.

bunzip2 Descompacta arquivos bz2.

bzip2recover Recupera arquivos bz2 extragados.

```
-t          Testa
-v          Modo verbose,
```

bz2cat Descompacta para tela (stdout).

lha Cria e expande arquivos lharc.

unarj Descompacta arquivos arj.

split Útil para copiar um arquivo grande para disketes.
Gera os arquivos xaa,xab,xac,... Veja man split.

```
#Dividir um arquivo de 10mb em disketes de 1.44mb:
split -b 1440kb nomeArquivoGrande.doc
#Para recuperar use o cat:
cat xaa xab xac > nomeArquivoGrande.doc
```

tar O tar permite a você agrupar um conjunto de arquivos em um único, facilitando o backup (ou o envio pela internet). Pode-se ainda compactar os arquivos agrupados com o gzip.

```
-c          Cria novo arquivo.
-v          Modo verbose.
-z          Descompacta arquivos (*.tar.gz).
-f          NomeArquivo.
-w          Modo iterativo.
-r          Acrescenta no modo apende.
-x          Extrai arquivos.
-A         Concatena arquivos.
-d          Compara diferenças entre arquivos.
--delete   Deleta arquivos.
-t          Lista o conteúdo do arquivo.
-u          Atualiza o arquivo.
-N         Após a data tal.
```

```
-o      Extrai arquivos para monitor.
-w      Iterativa.
-C      Muda o diretório.
-G      Backup incremental.
```

```
#Empacotando origem em destino
tar -cf origem > destino.tar
#Empacotando e já compactando
tar -cvzf destino.tar.gz origem
#Desempacotando
tar -xf nome.tar
tar -xzvf nome.tgz
tar -xzvf nome.tar.gz
tar -xvzf nome.tar
#Backup completo no dispositivo /dev/fd0
tar cvfzM /dev/fd0 /          #Backup
tar xvfzM /dev/fd0           #Restauração
```

Diversos

[*] Metacaracter.

[?] Caracter coringa.

[a-c] Caracter coringa.

clear Limpa a tela.

date Mostra data e hora.

```
date -s "8:50"          #Acerta hora para 8:50
```

41.1.2 Expressões regulares³

Alguns programas do GNU-Linux aceitam o uso de expressões regulares (principalmente os da GNU). O uso de expressões regulares é útil nas pesquisas de arquivos com find, no uso do emacs, entre outros.

Alguns caracteres são usados para passar parâmetros para o interpretador das expressões regulares, para usá-los na pesquisa, deve-se preceder o caracter com /.

Veja abaixo alguns exemplos de expressões regulares.

[xyz] Qualquer das letras dentro do colchete.

[^xy] Exceto x e y.

[t-z] De t até z (tuvwxyz).

[a-zA-Z] Qualquer letra.

[0-9]	Qualquer número.
^	No início do parágrafo.
\$	No fim do parágrafo.
\<	No início da palavra.
\<search	Palavra que inicia com search.
\>	No fim da palavra.
\>search	Palavra que termina com search.
z*	Letra z, 0 ou mais vezes: z,zz,zzz,zzzz,...
Z+	Letra Z, 1 ou mais vezes.
A?	Letra A pode ou não fazer parte.
revistas?	(revista ou revistas).
A{m}	Letra A m vezes.
[0-9]{3}	Número de 3 dígitos.
Z{3,5}	Letra Z de 3 a 5 repetições zzz,zzzz,zzzzz
K{,7}	Até 7 repetições de K.
K{3,}	No mínimo 3 repetições de K.
{0,}	O mesmo que *.
{1,}	O mesmo que +.
{0,1}	O mesmo que ?.
()	Usado para deixar a expressão mais clara (precedências).

Linux-(6|6.1|6.2), Acha Linux-6, Linux-6.1, Linux-6.2.

O uso de () cria variáveis internas que podem ser acessadas como nos exemplos abaixo:

(quero)-\1 = quero-quero.

([a-zA-Z]\1 Qualquer letra espaço qualquer letra.

. Qualquer caracter . Se estiver no meio de uma sentença use \.

\w Qualquer letra.

\W Nenhuma letra.

| Pipe (tubo).

Conversão de wildcard (dos), para expressão regular.

```
*.txt      *\*.txt.
arq.cpp    Arq\*.cpp.
arq?.cpp   Arq\*.cpp.
Cap[1-7].lyx  Cap[1-7]\*.lyx.
arq{a,b}   arq(a|b).
```

Exemplo:

```
var=Avancado Define variável var, cujo conteúdo é o diretório Avancado.
              Para usar uma variável definida use $var.
ls var        Não aceita (ls: var: Arquivo ou diretório não encontrado).
ls $var       Mostra o conteúdo de var, do diretório Avancado.
ls 'var'      Não aceita aspas simples.
ls "echo $var" Não interpreta conteúdo de aspas duplas.
ls 'echo $var' Interpreta o conteúdo que esta dentro da crase.
              for i in *.html; do 'mv $i $i.old' ; done
```

41.1.3 Programas telnet e ftp

Apresenta-se a seguir os principais comandos dos programas telnet e ftp. Atualmente o telnet e o ftp estão sendo substituídos pelo ssh. O ssh é um secure shell, permitindo acessar, copiar e pegar arquivos de máquinas remotas. Isto é, o ssh funciona como o telnet e o ftp, mas com segurança.

telnet

O TELNET é um programa que é executado em seu computador e o conecta a outros computadores em qualquer lugar do mundo. É como se você estivesse executando os programas do computador remoto de dentro de seu computador.

Comando	Descrição.
?	Lista os comandos.
open	Conecta com um computador remoto.
display	Mostra os parâmetros da operação.
mode	Modo linha a linha ou caracter a caracter.
Set,	unset Seta os parâmetros de operação.

send	Transmit caracteres especiais.
status	Informações de estatus.
Contrl +z	Suspende o telnet, chama o shell.
fg	Retorna do shell para o telnet. Volta para o último programa em operação.
logout	Encerra conexão avisando.
close	Fecha a conexão corrente.
quit	Encerra o telnet.

ftp

O FTP precisa de 2 computadores, um cliente e outro servidor. O programa FTP cliente é executado em seu computador e o programa FTP servidor em um computador remoto. O cliente envia comandos (como listar diretórios) para o servidor, o servidor responde (mostrando o diretório).

Depois de localizado o arquivo (que você deseja baixar), o mesmo pode ser obtido com o comando `get nome_do_arquivo`. Com `quit` você encerra a conexão FTP.

help	Lista de todos os comandos ftp.
!	Pausa o ftp e inicia o shell.
! comando	Executa um comando do shell.

Conectando

open h	Inicia conexão com host h.
user	Define usuário.
ascii	Transferência de arquivo no formato ascii.
binary	Transferência de arquivo no formato binário.
hash yes/no	Mostra o # do arquivo transferido.
prompt yes/no	Aguarda resposta para transferência de múltiplos arquivos.
status	Exibe configuração atual.
get	Puxa o arquivo para seu computador.
mget	Puxa diversos arquivos.
send	Envia um arquivo (ou put, ou mput).
cd nome	Muda o diretório.
cdup	Diretório pai.

<code>dir</code> ou <code>ls</code>	Mostra diretório com detalhes.
<code>lcd</code>	Muda diretório local.
<code>pwd</code>	Mostra a path corrente.
<code>quit</code>	Encerra a conexão.
<code>close</code>	Encerra conexão.

Agora você já conhece os comandos e programas mais utilizados em um terminal do Linux. Podendo navegar, criar, deletar e mover diretórios. Procurar e visualizar o conteúdo de arquivos.

Como dito anteriormente, você pode consultar o manual de sua distribuição para obter maiores informações sobre estes comandos.

41.2 Diretórios úteis para programadores

Quem programa no Linux precisa saber onde estão o diretório com os programas do sistema, o diretório onde são instalados os programas e onde estão as bibliotecas. Veja na Tabela 41.1 alguns destes diretórios.

Tabela 41.1: Diretórios importantes para o programador.

Diretório	Descrição
<code>/usr/bin</code>	Programas do sistema.
<code>/usr/local/bin</code>	Programas locais estão aqui.
<code>/usr/include</code>	Arquivos include
<code>/usr/lib</code>	Bibliotecas
<code>/usr/openwin/lib</code>	Bibliotecas do X window

41.3 Programas úteis para programadores

Apresenta-se na Tabela 41.2 uma lista de programas úteis usados no desenvolvimento de programas no Linux. Estes programas serão detalhados posteriormente.

Tabela 41.2: Programas úteis para desenvolvedores de software no ambiente Linux.

	Programas utilitários	
	Ambientes de desenvolvimento	
	Ambiente para desenvolvimento no KDE	kdevelop
	Ambiente de desenvolvimento semelhante ao Borland	xwpe
	Ambiente para desenvolvimento no GNOME	glade
	Editor de texto	
	Editor de texto simples	emacs
	Compiladores	
	Compilador C da GNU	gcc
	Compilador C++ da GNU	g++
	Compilação automatizada	make
	Linkagem	ld
	Depuração	
	Depurador C da GNU	gdb
	Depurador do KDE (kdevelop)	kdbg
	Programas GNU Compliant	
	Geração dos scripts de configuração	autoconf
	Geração dos scripts Makefile	automake
	Pacote de geração de bibliotecas	libtool
	Programas Diversos	
	Traça informações, chamadas a bibliotecas	ltrace
	Controle de versões	CVS
	Formatação do código	
	Diferenças entre arquivos	diff
	Formata o código	bison
	Organiza o código (layout)	indent
	Analisador léxico	flex,flex++
	Documentação	
	Gera documentação a partir de tags no código.	doxygen
	Geração de diagramas de classes	graphviz
	Manipulação binária	bfd
	Binary file descriptor	binutil
	Profile (analisador de desempenho)	gprof
	Montagem de bibliotecas	ar
	Índices das bibliotecas	ranlib
	Informações sobre objetos	objdump

Capítulo 42

Edição de Texto Emacs e VI

O emacs, ilustrado na Figura 42.1 é um bom editor de texto. O emacs oferece um conjunto de funções específicas para o desenvolvimento de seus programas em C++.

Ao salvar o arquivo com a extensão *.h ou *.cpp, automaticamente o emacs mostra a sintaxe de C++ através do “syntax-highlight”, isto é, o texto aparece colorido.

Para maiores informações sobre o emacs procure no <http://www.altavista.com> por *emacs programming*.

Se você gosta de outros editores de texto, não tem problema. Use o que você conhece.

42.1 Comandos do editor emacs

Lista-se a seguir, de forma resumida os principais comandos do emacs. A letra ^ representa o CTRL.

42.1.1 Help

^+h n	Emacs news.
^h i	Info.
^h m	Modo de descrição.
^h a	Comando apropos.
^h t	Emacs tutorial.
^h f	Descrição da função.
C-x	Acesso a função de um único caracter.
M-x	Acesso a comando de uma palavra.

42.1.2 Movimento do cursor (use as setas de direção)

Alt+<	Início do arquivo.
Home	Início do arquivo.

pageUp Início da página.
^p Início do parágrafo.
^b ^f Linhas.
^a ^e Fim do parágrafo.
Alt+b Alt+f Palavras.
^n
Alt+> Fim do arquivo.
pageDown Fim da página.
End Fim do arquivo.

42.1.3 Cut/Copy/Paste/Undo

Alt+w Copy.
^w Cut.
^y Paste.
^x u Undo.
^_ Undo.
Alt+y Paste sucessivo.
^d Apaga a letra a direita.
del Apaga a letra a esquerda.
Alt+d Apaga a palavra a direita.
Alt+del Apaga a palavra a esquerda.
^k Apaga toda a linha a direita.
Alt+k Apaga toda a linha a direita inclusive retorno de carro.

42.1.4 Arquivos

^x ^f Abre um arquivo do disco ou cria novo.
^x ^d Abre o diretório.
^x ^s Salva o arquivo em disco.
^x ^w Salvar como.
^x ^d Abre um diretório.

<code>^x 1</code>	Uma janela.
<code>^x 2</code>	Duas Janelas.
<code>^x i</code>	Inserir o arquivo.
<code>^x ^b</code>	Lista os buffers.

42.1.5 Pesquisa e substituição

<code>Alt+%</code>	Entre com a string A Entre com a string B #Pede confirmação
--------------------	--

<code>Alt+x repl s</code>	Entre com a string A Entre com a string B #Não pede confirmação
---------------------------	--

Pesquisa

<code>^s palavra</code>	Procura pela palavra (para baixo).
<code>^r palavra</code>	Procura pela palavra (para cima).

42.1.6 Múltiplas janelas

<code>^u 0^]</code>	Posiciona no inicio da janela
<code>^mv</code>	Scroll para final da janela
<code>^xo</code>	Alterna janela ativa
<code>^x2</code>	Duas janelas
<code>^x1</code>	Uma janela ativa

42.1.7 Encerrando seção do Emacs

<code>^x ^c</code>	Sai do Emacs.
<code>^z</code>	Suspende o Emacs.

Sentenças para o emacs

- AUTO SAVE: O emacs salva automaticamente os arquivos em disco com o nome: "#nomeArquivo#". Quando voce salva o arquivo em disco deixa de existir o autoSave.
- A Linha de informações mostra: O nome do arquivo, a linha, a porcentagem

42.2 Comando do editor vi

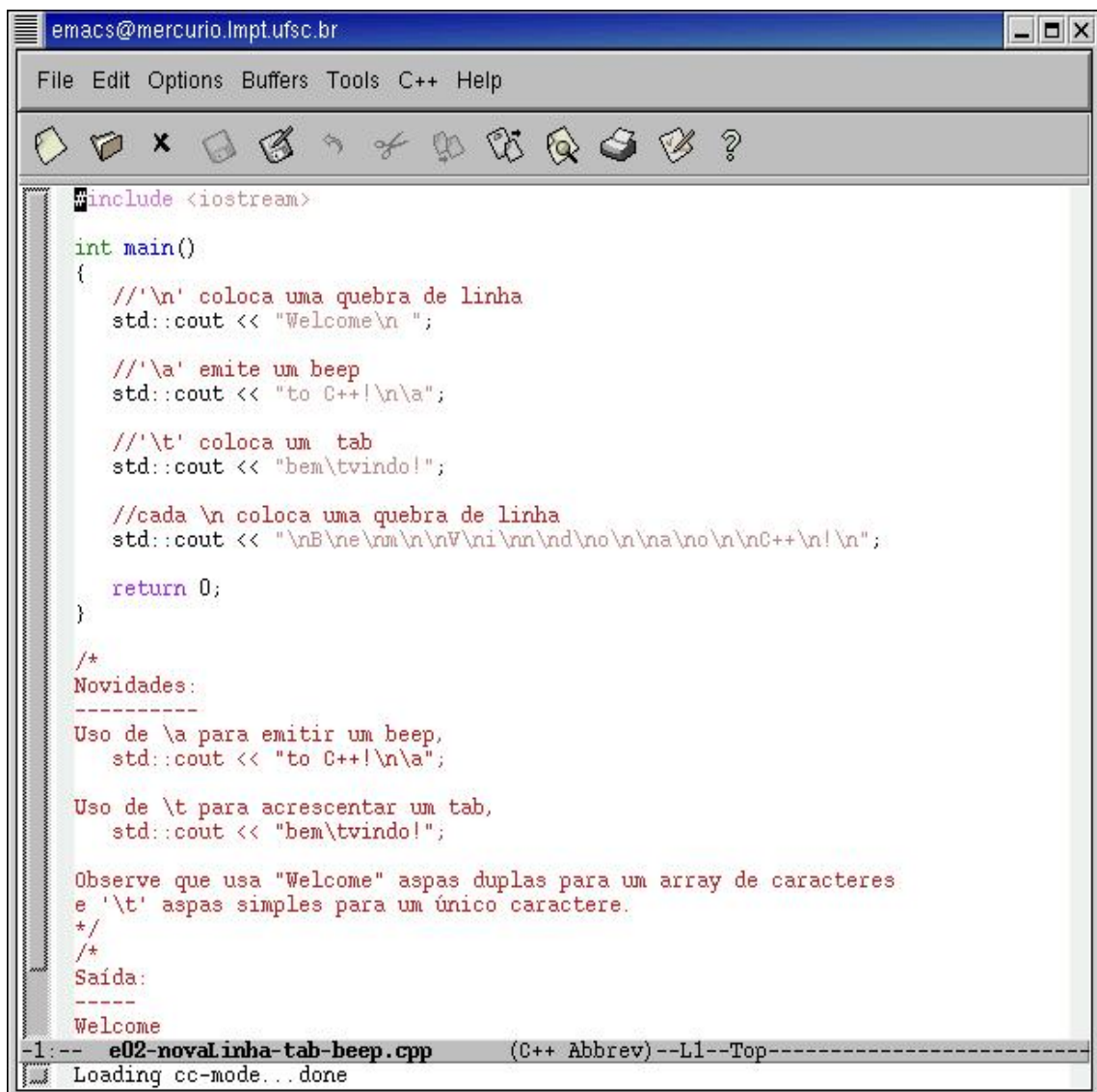
Editor de texto simples e eficiente.

```
:w          Salva arquivo em disco.  
:q          Para sair.  
:q!         Sai mesmo que o arquivo tenha sido alterado.  
:e          Edita outro arquivo.  
:!com       Executa comando do shell.  
:r arq      Lê outro arquivo na posição do cursor.
```

Exemplo:

```
vi arquivo #Edita o arquivo  
man vi     #Maiores informações sobre o vi
```

Figura 42.1: O editor de texto emacs.



The image shows a screenshot of the Emacs text editor window. The title bar reads "emacs@mercurio.lmpt.ufsc.br". The menu bar includes "File Edit Options Buffers Tools C++ Help". The toolbar contains icons for file operations and editing. The main text area displays a C++ program that demonstrates the use of escape sequences: '\n' for a new line, '\a' for a beep, and '\t' for a tab. The program's output is shown at the bottom of the window.

```
emacs@mercurio.lmpt.ufsc.br
File Edit Options Buffers Tools C++ Help
[Icons]
#include <iostream>

int main()
{
    //'n' coloca uma quebra de linha
    std::cout << "Welcome\n ";

    //'a' emite um beep
    std::cout << "to C++!\n\a";

    //'t' coloca um tab
    std::cout << "bem\tvindo!";

    //cada \n coloca uma quebra de linha
    std::cout << "\nB\nC\nD\nE\nF\nG\nH\nI\nJ\nK\nL\nM\nN\nO\nP\nQ\nR\nS\nT\nU\nV\nW\nX\nY\nZ\n[\\n\n]\n";

    return 0;
}

/*
Novidades:
-----
Uso de \a para emitir um beep,
    std::cout << "to C++!\n\a";

Uso de \t para acrescentar um tab,
    std::cout << "bem\tvindo!";

Observe que usa "Welcome" aspas duplas para um array de caracteres
e '\t' aspas simples para um único caractere.
*/
Saída:
-----
Welcome
-1:-- e02-novaLinha-tab-beep.cpp (C++ Abbrev)--L1--Top-----
[Icons] Loading cc-mode... done
```


Capítulo 43

Os programas diff, patch, indent

Apresenta-se neste capítulo uma breve introdução aos programas diff, patch e indent. Este capítulo pode ser lido mais tarde sem perda de sequência.

43.1 O programa diff

O programa diff é usado para mostrar as diferenças entre 2 arquivos, isto é, compara dois arquivos linha a linha.

O programa diff é muito útil, o mesmo é usado para comparar versões de um mesmo arquivo (veja Capítulo 54) e para gerar arquivos de patch (veja seção 50.1.3).

Protótipo e parâmetros do diff:

diff [opções] Arq1 Arq2

- b* Ignora espaços em branco.
- c* Gera saída mostrando os 2 arquivos e as diferenças.
- i* Ignora diferenças entre maiúsculas e minúsculas.
- q* Apenas informa se os arquivos são iguais ou diferentes.
- r* Compara diretórios recursivamente.
- v* Mostra versão do diff.
- x* *pattern* Quando compara diretórios, considerar arquivos com a extensão *pattern*.
- u* Formato unificado (+ claro).

Veja a seguir o arquivo e06a-hello.cpp, um programa simples em C++.

Listing 43.1: Arquivo e06a-hello.cpp.

```
#include <iostream>

int main()
{
    std::cout << "Oi_tudo_bem_"<<std::endl;
    return 0;
}
```

O arquivo e06a-hello.cpp foi modificado e salvo como e06b-hello.cpp. Veja a seguir o arquivo e06b-hello.cpp, observe que foram acrescentadas 4 linhas novas e o return foi alinhado.

Listing 43.2: Arquivo e06b-hello.cpp.

```
#include <iostream>

int main()
{
    std::cout << "Oi tudo bem " << std::endl;
    std::cout << "Entre com x " << std::endl;
    int x;
    cin >> x;
    std::cout << "x = " << x << std::endl;

    return 0;
}
```

Veja a seguir o arquivo gerado pelo comando: **diff e06a-hello.cpp e06b-hello.cpp**. O símbolo < indica que esta linha saiu (é velha). O símbolo > indica que esta linha entrou (é nova).

Listing 43.3: Arquivo diff.

```
6c6,11
<     return 0;
---
>     std::cout << "Entre com x " << std::endl;
>     int x;
>     cin >> x;
>     std::cout << "x = " << x << std::endl;
>
>     return 0;
```

Oberve a linha 6c6,11. A mesma significa que a linha 6 do arquivo e06a-hello.cpp e a linha 6 do arquivo e06b-hello.cpp são diferentes, existe um espaço extra no arquivo e06a-hello.cpp. O **c** indica modificado (changed).

Podem aparecer as letras **a** de adicionado, **c** de modificado (changed) e **d** de deletado.

Veja a seguir o arquivo gerado pelo comando: **diff -c e06a-hello.cpp e06b-hello.cpp**. Observe que os dois arquivos são mostrados na íntegra. O caracter ! indica as linhas que foram alteradas.

Listing 43.4: Arquivo diff -c.

```
*** e06a-hello.cpp      Tue Jun  4 13:23:49 2002
--- e06b-hello.cpp      Tue Jun  4 13:24:46 2002
*****
*** 3,8 ****
    int main()
    {
        std::cout << "Oi tudo bem " << std::endl;
!     return 0;
    }

--- 3,13 ----
    int main()
    {
        std::cout << "Oi tudo bem " << std::endl;
```

```
!   std::cout << "Entre com x " << std::endl;
!   int x;
!   cin >> x;
!   std::cout << "x= " << x << std::endl;
!
!   return 0;
}
```

Veja a seguir o arquivo gerado pelo comando: **diff -u e06a-hello.cpp e06b-hello.cpp**. No início do arquivo a nomenclatura — se refere ao arquivo **e06a-hello.cpp** e **+++** ao arquivo **e06b-hello.cpp**.

Observe que no formato **-u** aparecem todas as linhas precedidas por um sinal **+/-**. O sinal **-** indica que a linha saiu e o sinal **+** que entrou.

Listing 43.5: Arquivo diff -u.

```
--- e06a-hello.cpp      Tue Jun  4 13:23:49 2002
+++ e06b-hello.cpp      Tue Jun  4 13:24:46 2002
@@ -3,6 +3,11 @@
     int main()
     {
         std::cout << "Oi tudo bem " << std::endl;
-        return 0;
+        std::cout << "Entre com x " << std::endl;
+        int x;
+        cin >> x;
+        std::cout << "x= " << x << std::endl;
+
+        return 0;
     }
```

43.1.1 Sentenças para o diff

- O formato gerado pelo diff **-u** é o mais claro.
- Você pode gerar um arquivo com as diferenças.

Exemplo:

```
diff e06a-hello.cpp e06b-hello.cpp > diferencas.txt
```

- O formato padrão gerado pelo diff é usado pelo programa patch.
- Existem outros programas para comparar arquivos, veja nas páginas man de seu Linux os programas: **diff3** (compara 3 arquivos), **wdiff**, **mp**, **sdiff**.
- Ao editar, a saída do comando **diff** no programa **emacs**, o mesmo aparece com sintaxe especial.

43.2 O programa patch¹

O programa **path** é usado para unificar arquivos.

¹Veja na seção 50.1.3 o uso do programa **diff** para distribuição de upgrades de programas.

Protótipo e parâmetros do patch:

patch [-u-v] arquivoAntigo arquivoDiferencas.

- u *O arquivo das diferenças foi gerado usando a versão unificada (opção -u no diff).*
- v *Mostra versão do programa patch.*

Dados os arquivos arq1.cpp, arq2.cpp e o arquivo das diferenças gerado pelo diff, isto é, você pode atualizar o arquivo arq1.cpp de forma que o mesmo tenha o mesmo conteúdo do arq2.cpp. Veja o exemplo.

Exemplo:

```
diff arq1 arq2 > diferencas.cpp
//Apenas por segurança crie um backup do arq1
cp arq1 arq1.backup
//Vai modificar o arq1 e o mesmo ficará igual a arq2
patch arq1.cpp diferencas.cpp
```

43.3 O programa indent

O programa indent é usado para deixar o código organizado, através do uso de padrões de indentação. Existe um padrão default, pré-estabelecido, mas você mesmo pode defini-los padrões a serem utilizados. Veja a seguir o protótipo de uso do indent. Lista-se, de forma abreviada, alguns dos parâmetros que podem ser passados para o indent. Estes parâmetros podem ser armazenados no arquivo “.indent.pro”.

Protótipo:

```
indent file [-o outfile ] [ opções ]
indent file1 file2 ... fileN [ opções ]
```

- st Envia saída para tela.
- gnu Usa opções de formatação da GNU.
- orig Usa opções de formatação da Berkeley.
- v Modo verbose.
- l60 Limita a 60 colunas.
- bad Linha em branco após declarações (Para desativar, -nbad).
- bap Linha em branco após definições de funções (-nbap).
- bbb Linha em branco antes de uma caixa de comentário.
- sc Comentários no estilo de C /* * */.
- bl5 Colchetes do bloco alinhados a 5 caracteres.
- bn Bloco alinhado.
- bli5 Bloco alinhado com 5 espaços.
- bls Alinha o par{ }.

- cli2 Bloco switch alinhado com espaço 2.
- npcs Sem espaço entre o nome da função e o ().
- cs Espaço depois do cast.
- di16 Indenta nome dos objetos em 16 espaços.
- bfda Quebra argumentos da função em várias linhas.
- lp Alinha parâmetros de funções com nomes grandes.

Dica: No código você pode desabilitar ou habilitar o indent.

```
/* * indent_on */,  
/* * indent_off */.
```

O programa *ex-vector-1.cpp* apresentado na seção 6.4, foi modificado com o programa indent com o comando:

Exemplo:

```
cp ex-vector-1.cpp ex-vector-1-indent.cpp  
indent ex-vector-1-indent.cpp
```

veja a seguir a listagem do arquivo *ex-vector-1-indent.cpp*. Compare esta listagem com a listagem 6.4. Observe a mudança na declaração da função *main*, na forma do *do..while* e nas indentações.

Listing 43.6: Arquivo *ex-vector-1-indent.cpp*.

```
//Classes para entrada e saída  
#include <iostream>  
  
//Classe pra formatação de entrada e saída  
#include <iomanip>  
  
//Classe de vetores, do container vector  
#include <vector>  
  
//Classe para algoritimos genéricos  
//#include <algorithm>  
  
//Define estar usando espaço de nomes std  
using namespace std;  
  
//Definição da função main  
int  
main ()  
{  
    //Cria vector, do tipo int, com nome v  
    vector < int >v;  
  
    int data;  
    cout << "No_DOS_ um_ctrl+d_encerra_a_entrada_de_dados." << endl;  
    cout << "No_Mac_ um_ctrl+d_encerra_a_entrada_de_dados." << endl;  
    cout << "No_Linux_ um_ctrl+d_encerra_a_entrada_de_dados." << endl;  
    do  
    {
```

```

    cout << "\nEntre com o dado (" << setw (3) << v.size () << "):";
    cin >> data;
    cin.get ();
    //acidiona ao final do vetor o objeto data
    v.push_back (data);
}
while (cin.good ());

//Acessa partes do vector usando funções front e back
cout << "\nPrimeiro elemento do vetor=" << v.front ()
    << "\nÚltimo elemento do vetor=" << v.back () << endl;

//mostra o vetor
for (int i = 0; i < v.size (); i++)
{
    cout << "v[" << setw (3) << i << "]= " << setw (5) << v[i] << ' ';
}
cout << endl;

cout << (v.empty ()? "vazio" : "não vazio") << endl;

//Chama função clear
v.clear ();
cout << (v.empty ()? "vazio" : "não vazio") << endl;

cout << endl;
cin.get ();
return 0;
}

/*
Novidade:
Uso do container vector.
Uso das funções: push_back, size, empty, clear,
*/

```

Sentenças para o indent

- Opções padrões do padrão -gnut
 - -nbad -bap -nbc -bbo -bl -bli2 -bls -ncdb -nce -cp1 -cs -di2 -ndj -nfc1 -nfca -hnl -i2 -ip5 -lp -pcs -nprs -psl -saf -sai -saw -nsc -nsob
- Para maiores informações sobre o indent consulte as informações do programa usando **info indent**.
- Para ver todas as opções do indent use: **man indent**.

Capítulo 44

Compilando com gcc, g++

O gcc/g++ é o compilador C/C++ da GNU. Pode ser baixado no site da gnu, individualmente, ou como parte do pacote do EGCS (que inclui um conjunto de programas acessórios).

Dica: Para conhecer em detalhes o gcc baixe o manual do gcc no site da gnu.

44.1 Protótipo e parâmetros do gcc/g++

Apresenta-se aqui o protótipo e os parâmetros que você pode passar para o compilador da gnu. Posteriormente, através dos exemplos, você compreenderá melhor cada parâmetro.

Protótipo do gcc/g++:

g++ [opções] [parâmetros] arquivos.

- v* *Mostra detalhes da compilação.*
- wall* *Mostra todos os warnings.*
- onome* *Define o nome do arquivo de saída (opcional).*
- w* *Elimina mensagens de warning.*
- I/path* *Acrescenta path include.*
- l/path/lib* *Inclue biblioteca (lib).*
- ggdb* *Informações extras para o gdb.*
- O* *Optimiza o código (-O1,-O2,-O3).*
- c* *Somente compila (gera o arquivo *.o).*
- S* *Somente compila o arquivo, não linka.*
- lcomplex* *Inclue biblioteca dos complexos.*
- lm* *Inclue biblioteca matemática.*
- E* *Cria somente o arquivo pré-processado.*
- C* *Não inclue comentários no executável.*
- g* *Gera informações para o debugger (código lento).*

- qdigraph* Adiciona teclas dígrafas.
- qcompact* Deixa o código mais compacto.
- xlinguagem* Especifica a linguagem (C, C++, assembler).
- p* Informações para o profiler *proff*.
- pg* Informações para o *groff*.
- m686* Especifica que a máquina alvo é um 686.
- static* Especifica que a linkagem deve ser estática.
- p* Especifica inclusão de instruções para o profiler.
- pg* Especifica inclusão de instruções para o profiler da *gnu* (*gprof*).

44.2 Arquivos gerados pelo gcc/g++

A medida que os arquivos são compilados e linkados, são gerados alguns arquivos adicionais. Lista-se a seguir os arquivos de entrada (*.h, *.cpp), e os arquivos de saída gerados pelo g++.

- *.h Arquivos header.
- *.i Arquivos de pré-processamento para programas em C.
- *.ii Arquivos de pré-processamento para programas em C++.
- .c,.cc,.C,.c++,.cpp, Arquivos de fonte.
- .o Arquivo objeto.
- .s Arquivo assembler.
- .a Arquivo de biblioteca estática.
- .sa Blocos de bibliotecas estáticas linkados ao programa.
- .so.n Arquivo de biblioteca dinâmica.
- a.out Arquivo de saída (executável).

44.3 Exemplo de uso do gcc/g++

Apresenta-se a seguir um exemplo básico.

1. Edita o programa `hello.cpp` com o `emacs`.
Abra um terminal e execute **emacs hello.cpp**.


```
#include <iostream>
using namespace std;
main()
{
cout << "hello!" << endl;
}
/*No emacs use ctrl+x ctrl+s para salvar o arquivo
e ctrl+x ctrl+q para sair do emacs*/
```

2. Cria o arquivo de pré-processamento [opcional]

```
g++ -E hello.cpp
```

3. Compila o módulo hello (gera o hello.o)

```
g++ -c hello.cpp
```

4. Linka o programa e gera o executável

```
g++ -ohello hello.o
```

5. Executa o programa

```
./hello
```

6. Observe que os passos 2,3 e 4 podem ser executados usando:

```
g++ -v -ohello hello.cpp
```

O -v mostra um conjunto de informações a respeito dos passos da compilação.

Neste exemplo o nome do programa executável é hello e foi especificado com a opção -ohello. Se você não especificar um nome, o programa executável se chamará **a.out**.

Tarefa: Compare o tamanho dos códigos (dos executáveis) gerados com as opções:

```
g++ hello.cpp          #comun
g++ -g2 hello.cpp      #com debugger
g++ -o3 hello.cpp      #com otimização
```


Capítulo 45

Make

No capítulo anterior você aprendeu a usar o g++ para compilar um programa, como você pode constatar, é fácil. Mas se seu programa for grande, você vai ter de executar o g++ para cada arquivo *.cpp. O que se transforma em trabalho tedioso.

Para evitar este trabalho, foram desenvolvidos os arquivos de projeto. Neste capítulo vamos descrever o que é, como escrever, e como usar arquivos de projeto e o programa make.

45.1 Um arquivo de projeto

Um arquivo de projeto permite a compilação de diversos arquivos com uso do programa make.

- Você pode compilar seus programas diretamente (digitando a instrução de compilação), ou usando arquivos makefile.
- Um arquivo de projeto pode ser simples ou complexo, vai depender do tamanho do programa e dos recursos utilizados.
- Um arquivo de projeto do Borland C++ e do MFC tem um formato proprietário, que só pode ser lido por estes programas. Um arquivo de projeto do kdevelop¹ é um arquivo ASCII, podendo ser editado em qualquer editor simples.
- Um arquivo makefile é um arquivo de projeto no formato ASCII, que pode ser editado em editores simples, como o emacs.
- Por padrão, um arquivo de projeto makefile tem o nome **makefile** ou **Makefile**.

Bem, então podemos criar um arquivo Makefile, um arquivo com instruções de compilação. Estas instruções de compilação serão lidas e executadas pelo programa make. Ou seja, o make lê o arquivo Makefile e executa as tarefas ali descritas, automatizando a compilação de programas complexos. Descreve-se a seguir como funciona o programa make.

¹Se você gosta de interface amigável, pode usar o excelente ambiente de desenvolvimento do kdevelop. Um ambiente de desenvolvimento completo e com interface amigável.

45.2 Protótipo e parâmetros do make

Apresenta-se a seguir o protótipo e os parâmetros do make. A opção `-f` é usada para passar o nome do arquivo makefile. Por padrão, o make procura no diretório corrente o arquivo makefile e depois o arquivo Makefile.

Protótipo do make:

```
make [ -f arq_makefile ] [ opções ] ... alvos ...
```

- e* *Indica variáveis do ambiente que devem prevalecer sobre atribuições feitas no make.*
- k* *Desconsiderar erros.*
- n* *Apenas lista os comandos, sem executá-los.*
- p* *Imprime alguns resultados.*
- r* *Despreza regras intrínsecas.*
- t* *Atualiza data e hora.*
- s* *Modo silencioso.*
- f* *arq_makefile Especifica o nome do arquivo makefile.*

45.3 Formato de um arquivo Makefile

Um arquivo makefile contém um conjunto de instruções que são lidas e executadas pelo programa make. Basicamente, um arquivo makefile é dividido em uma parte com definições de variáveis e outra parte com sub-rotinas a serem executadas.

A primeira parte define variáveis a serem utilizadas. As variáveis incluem o nome do compilador, as paths de inclusão de arquivos e bibliotecas, e listagens de arquivos a serem processados.

45.3.1 Criando variáveis em um arquivo Makefile

O make aceita que você defina variáveis em seu arquivo makefile e depois use estas variáveis nas etapas de compilação.

No exemplo a seguir cria uma variável e usa seu conteúdo. Observe que para obter o valor da variável você deve usar um `$()`. Ao usar `$(VARIABLE)`, o make substitue o nome da variável pelo seu valor.

Exemplo:

```
NOMEVARIABLE = valor
$(NAMEVARIABLE)
```

- Por convenção, as variáveis definidas em um arquivo makefile são todas maiúsculas.
- Você pode criar variáveis da forma

```
NOME_DA_VARIABLE
CC = g++
```

- E a seguir modificar seu valor, acrescentando algo em modo append

```
CC += -O2
```

- ²Make usa um conjunto de variáveis com nomes pré-definidos.

AR	Especifica o programa de manutenção de arquivos.
CC	Especifica o compilador, default=cc.
CPP	Especifica o pré-processador C++.
RM	Programa de remoção de arquivos, default = rm -f .
CFLAGS	Flags que devem ser passados para o compilador C.
CPPFLAGS	Flags que devem ser passados para o compilador C++.
LDFLAGS	Flags que devem ser passados para o linker.

45.3.2 Criando alvos em um arquivo Makefile

A segunda parte de um arquivo makefile contém alvos a serem executados. O formato padrão para um alvo é dado por:

Protótipo de um alvo:

alvo: Dependências
Instruções a serem executadas

- As instruções a serem executadas iniciam com um tab (e não 8 espaços).
- Geralmente o alvo é o nome de um arquivo, uma biblioteca ou um programa a ser compilado.
- Alvos usuais em arquivos makefile são:

all	Executar todas as dependências.
install	Instalar o programa.
uninstal	Desinstalar o programa.
dist	Gerar uma distribuição no formato .tar.gz.
check	Verificar a consistência da instalação.

45.4 Exemplo de um arquivo Makefile

Veja na listagem a seguir um exemplo de arquivo makefile.

Listing 45.1: Arquivo makefile.

```
#Toda linha começada com # é uma linha de comentário
#-----
#Parte I: Definição de variáveis
ARQUIVOS=e06a-hello.cpp
OBJETOS=e06a-hello.o
```

```

DIRINCLUDE = -I/usr/include/g++ -I/usr/include
DIRETORIOLIB =
PARAMETROSLIB = -lm
COMPILADOR = g++

#-----
#Parte II: alvos
#all é o alvo, e06a-hello.o e e06a-hello são as dependências

all : e06a-hello.o e06a-hello

#e06a-hello.o é o alvo, $( ARQUIVOS) são as dependências
#e $( COMPILADOR) .. é a instrução
e06a-hello.o : $(ARQUIVOS)
    $(COMPILADOR) -c $(ARQUIVOS)  $(DIRINCLUDE) $(DIRECL) -o e06a-hello.o

e06a-hello : $(OBJETOS)
    $(COMPILADOR) $(OBJETOS) $(DIRINCLUDE) $(DIRETORIOLIB) $(PARAMETROSLIB)
    -o e06a-hello

clean :
    rm -f *.o *.obj

```

Veja na listagem a seguir uma sequência de execução do programa make usando o arquivo makefile-hello.

Listing 45.2: Exemplo de uso do programa make.

```

//Limpa os arquivos anteriores
[andre@mercurio Cap-GNU]$ make clean
rm -f *.o *.obj

//Compila o arquivo e06a-hello.o
[andre@mercurio Cap-GNU]$ make e06a-hello.o
g++ -c e06a-hello.cpp -I/usr/include/g++ -I/usr/include -o e06a-hello.o

//Gera o programa executável
[andre@mercurio Cap-GNU]$ make e06a-hello
g++ e06a-hello.cpp -I/usr/include/g++ -I/usr/include -lm -o e06a-hello

//Veja abaixo que o make não compila novamente o que esta atualizado
[andre@mercurio Cap-GNU]$ make e06a-hello
make: 'e06a-hello' está atualizado.

//Limpando os arquivos obj
[andre@mercurio Cap-GNU]$ make clean
rm -f *.o *.obj

//Observe abaixo que o alvo e06a-hello chama o alvo e06a-hello.o
[andre@mercurio Cap-GNU]$ make e06a-hello
g++ -c e06a-hello.cpp -I/usr/include/g++ -I/usr/include -o e06a-hello.o
g++ e06a-hello.cpp -I/usr/include/g++ -I/usr/include -lm -o e06a-hello

Dica: Observe a forma como os alvos foram traduzidos.
0 alvo:
e06a-hello : $(ARQUIVOS) $(OBJETOS)

```

```
$(COMPILADOR) $(ARQUIVOS) $(DIRETORIOINCLUDE) $(DIRETORIOLIB) $(  
PARAMETROSLIB) -o e06a-hello
```

Foi traduzido da forma:

```
g++ e06a-hello.cpp -I/usr/include/g++ -I/usr/include -lm -o e06a-hello
```

45.5 Sentenças para o make

- Para conhecer em detalhes o make baixe o manual do make no site da gnu (<http://www.gnu.org>).
- Os arquivos especificados nas dependências devem existir. Se não existirem vai acusar erro.
- Os arquivos de cabeçalho *.h também devem ser incluídos nas dependências. Isto evita a mensagem de erro do compilador pela falta dos mesmos.
- Make é inteligente, se você pediu para executar o alvo 2 e este depende do alvo 1, o make executa primeiro o alvo 1 e depois o alvo 2.
- Se alguma dependência sofre modificações, o make recompila os arquivos que foram modificados.
- Alvos sem dependência não são automaticamente executados.
- Ao editar um arquivo makefile ou Makefile no emacs, o mesmo aparece com sintaxe especial. Auxiliando a implementação do arquivo makefile.

Capítulo 46

Bibliotecas

Apresenta-se neste capítulo um conjunto de programas auxiliares, que são utilizados para montagens de bibliotecas no mundo Linux. A seguir apresenta-se um exemplo de montagem de biblioteca estática e um exemplo de montagem de biblioteca dinâmica.

46.1 Introdução a montagem de bibliotecas

O Linux tem um conjunto de programas auxiliares que podem ser utilizados para montagem de bibliotecas estáticas e dinâmicas. Os mesmos são utilizados para criar, manter e gerenciar bibliotecas. Apresenta-se a seguir uma breve descrição destes programas.

Para obter informações detalhadas de cada programa dê uma olhada no *man page* ou nos manuais dos programas (os manuais podem ser baixados no site da gnu (<http://www.gnu.org>)).

Uma biblioteca é uma coleção de objetos (funções, classes, objetos), agrupados em um único arquivo. De um modo geral, um conjunto de arquivos com a extensão *.o, são reunidos para gerar um arquivo libNome.a (para biblioteca estática) ou libNome.so (para biblioteca dinâmica).

46.1.1 ar

O programa ar é utilizado para manipular arquivos em um formato bem estruturado. O ar também cria tabelas com símbolos e referências cruzadas. O programa ar é que aglutina todos os objetos em uma lib, isto é, agrupa os arquivos *.o em uma lib. Veja a seguir o protótipo e um exemplo de uso do ar.

Protótipo e parâmetros do ar:

ar [opções] arquivos.

- t* *Lista os objetos da lib (biblioteca).*
- r* *Substitue funções quando necessário (arquivos antigos).*
- q* *Adiciona no modo append.*
- s* *Atualiza a tabela de símbolos.*
- c* *Cria o arquivo se este não existe.*
- v* *Modo verbose.*

Exemplo:

```
ar cru libNome.a arq1.o arq2.o arq3.o
```

Neste exemplo o programa ar vai juntar os arquivos arq1.o arq2.o arq3.o e gerar o arquivo libNome.a.

Veja a seguir a lista completa de opções do comando ar. A mesma pode ser obtida em seu sistema digitando **ar - -help**.

Listing 46.1: Saída do comando ar -help .

```
Usage: ar [-X32_64] [-]{dmpqrstx}[abcfilNoPsSuvV] [member-name] [count] archive
       -file file...
       ar -M [<mri-script>]
commands:
d       - delete file(s) from the archive
m[ab]  - move file(s) in the archive
p       - print file(s) found in the archive
q[f]   - quick append file(s) to the archive
r[ab][f][u] - replace existing or insert new file(s) into the archive
t       - display contents of archive
x[o]   - extract file(s) from the archive
command specific modifiers:
[a]     - put file(s) after [member-name]
[b]     - put file(s) before [member-name] (same as [i])
[N]     - use instance [count] of name
[f]     - truncate inserted file names
[P]     - use full path names when matching
[o]     - preserve original dates
[u]     - only replace files that are newer than current archive
contents
generic modifiers:
[c]     - do not warn if the library had to be created
[s]     - create an archive index (cf. ranlib)
[S]     - do not build a symbol table
[v]     - be verbose
[V]     - display the version number
[-X32_64] - (ignored)
Report bugs to bug-binutils@gnu.org and and hjl@lucon.org
```

46.1.2 ranlib

Gera os índices para a biblioteca, isto é, gera um mapa de símbolos que serão utilizados pelos programas para localizar corretamente as funções a serem executadas.

Protótipo e parâmetros do ranlib:

```
ranlib [-v -V] arquivo.
```

-v Versão do ranlib.

Exemplo:

```
ranlib libNome.a
```

46.1.3 nm

Mostra os símbolos da biblioteca.

Protótipo e parâmetros do nm:

nm [opções] arquivo.

-C /-demangle Mostra nomes de forma clara para o usuário.

-S /-print-armac Imprime índice dos símbolos.

Exemplo:

```
nm libNome.a
```

Veja a seguir a saída do comando **nm -help**.

Listing 46.2: Saída do comando nm -help .

Usage: nm [OPTION]... [FILE]...

List symbols from FILEs (a.out by default).

```
-a, --debug-syms      Display debugger-only symbols
-A, --print-file-name Print name of the input file before every symbol
-B                    Same as --format=bsd
-C, --demangle[={auto,gnu,lucid,arm,hp,edg,gnu-v3,java,gnat,compaq}]
                      Decode low-level symbol names into user-level names
  --no-demangle       Do not demangle low-level symbol names
  --demangler=<dso:function>
                      Set dso and demangler function
-D, --dynamic         Display dynamic symbols instead of normal symbols
  --defined-only      Display only defined symbols
-e                    (ignored)
-f, --format=FORMAT  Use the output format FORMAT.  FORMAT can be 'bsd',
                      'sysv' or 'posix'.  The default is 'bsd'
-g, --extern-only    Display only external symbols
-h, --help           Display this information
-l, --line-numbers   Use debugging information to find a filename and
                      line number for each symbol
-n, --numeric-sort   Sort symbols numerically by address
-o                    Same as -A
-p, --no-sort        Do not sort the symbols
-P, --portability    Same as --format=posix
-r, --reverse-sort   Reverse the sense of the sort
-s, --print-armac    Include index for symbols from archive members
  --size-sort        Sort symbols by size
-t, --radix=RADIX    Use RADIX for printing symbol values
  --target=BFDNAME   Specify the target object format as BFDNAME
-u, --undefined-only Display only undefined symbols
-V, --version        Display this program's version number
-X 32_64             (ignored)
```

```
nm: supported targets: elf32-i386 a.out-i386-linux efi-app-ia32 elf32-little
elf32-big srec symbolsrec tekhex binary ihex trad-core
Report bugs to bug-binutils@gnu.org and hjl@lucon.org.
```

46.1.4 objdump

Imprime informações sobre as bibliotecas e objetos.

Protótipo e parâmetros do objdump:

objdump [opções][parâmetros]

- d, -debugging.*
- syms Tabela de símbolos.*
- a Informações arquivo.*

Exemplo:

```
objdump -file-header file.o
```

Dica: Para maiores detalhes, execute **objdump - -help**.

46.1.5 ldd

Lista as bibliotecas dinâmicas que determinado programa usa.

Protótipo e parâmetros do ldd:

ldd [-d-r] programa

- -help Imprime um help.*
- -version imprime a versão do ldd.*
- d,- -data-relocs Processa uma realocação dos dados.*
- r,- -function-relocs Processa uma realocação dos dados e funções.*
- v,- -verbose Imprime informações em geral.*

Exemplo:

```
ldd /bin/netscape
```

Veja a seguir a saída do comando `ldd /usr/bin/lyx`. Lembre-se que LyX é o editor utilizado para montar esta apostila. Observe o uso das bibliotecas `libXForms`, `Xpm`, `X11` e `libstdc++`.

Listing 46.3: Saída do comando `ldd /usr/bin/lyx`.

```
libforms.so.0.88 => /usr/lib/libforms.so.0.88 (0x40032000)
libXpm.so.4 => /usr/X11R6/lib/libXpm.so.4 (0x400b4000)
libSM.so.6 => /usr/X11R6/lib/libSM.so.6 (0x400c3000)
libICE.so.6 => /usr/X11R6/lib/libICE.so.6 (0x400cc000)
libX11.so.6 => /usr/X11R6/lib/libX11.so.6 (0x400e3000)
libstdc++-libc6.2-2.so.3 => /usr/lib/libstdc++-libc6.2-2.so.3 (0
x401d9000)
libm.so.6 => /lib/i686/libm.so.6 (0x4021c000)
libc.so.6 => /lib/i686/libc.so.6 (0x4023f000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

46.1.6 ldconfig

O programa `ldconfig` determina os links necessários em tempo de execução para bibliotecas compartilhadas (shared libs)¹.

Protótipo e parâmetros do `ldconfig`:

```
ldconfig [-p-v]libs
```

```
-p      Mostra bibliotecas compartilhadas.
-v      Modo verbose.
```

46.2 Convenção de nomes para bibliotecas

- O nome de uma biblioteca deve iniciar com `lib`.
- A extensão será `*.a` para bibliotecas estáticas.
- A extensão será `*.so` para bibliotecas dinâmicas.
- Versões:
`libNome.so.VersãoMaior.VersãoMenor.Patch`
 A versão maior é uma versão incompatível com as demais.
 A versão menor inclui novidades.
 A patch inclui correções de bugs.
- Uma biblioteca que tem o nome encerrado com `-g` contém instruções de debugagem.
- Uma biblioteca que tem o nome encerrado com `-p` contém instruções para o profiler (`gprof`).

46.3 Bibliotecas usuais

Apresenta-se na Tabela 46.1 algumas bibliotecas usuais.

46.4 Montando uma biblioteca estática (`libNome.a`)

Para utilizar a biblioteca o programador precisa dos arquivos com o cabeçalho (formato como os objetos e as funções foram construídos) e o arquivo da `lib`.

Sequência para criar uma biblioteca estática no Linux:

1. Cria o arquivo de cabeçalho `*.h` (declaração das funções em C e das classes em C++)

```
emacs Tponto.h
Tponto.h class TPonto { .....};
//ctrl+x ctrl+s para salvar
//ctrl+x ctrl+c para sair
```

¹Descrito no livro *Linux Unleashed*, não disponível em minha máquina.

Tabela 46.1: Bibliotecas usuais.

Biblioteca	Uso
libGL.so	<GL/gl.h>, Interface para OpenGL
libjpeg.so	<jpeglib.h> Interface para arquivos jpeg
libpbm.so	<pbm.h> Interface para bitmaps monocromáticos
libpgm.so	<pgm.h> Interface para bitmaps tons de cinza
libpng.so	<png.h> Interface para arquivos portable bitmap format
libpnm.so	<pnm.h> Interface para bitmaps pbm, ppm, pgm
libpthread.so	<pthread.h> Posix Threads
libvga.so	<vga.h> Acesso a tela vga
libz.so	<zlib.h> Biblioteca para compactação de arquivos
glibc	Biblioteca padrão C
magick++	<magick++.h> Biblioteca gráfica.

2. Cria o arquivo de código *.cpp (definição das funções)

```
emacs TPonto.cpp
TPonto.cpp /*Define funções da classe*/ ....
```

3. Compila os arquivos de código (*.cpp) gerando os arquivos (*.o)

```
g++ -c TPonto.cpp
```

4. Cria a biblioteca (a lib)

```
ar -q libNome.a TPonto.o
```

5. Publica a biblioteca com

```
ranlib libTPonto.a
```

Observe que os itens 1 e 2 se referem a edição dos arquivos do programa. Em 3, o programa é compilado. Pode-se utilizar um arquivo makefile para automatizar esta tarefa. Em 4, cria-se a biblioteca, gerando-se o arquivo libTPonto.a. Em 5 a biblioteca é publicada no sistema.

46.4.1 Usando uma biblioteca estática

No exemplo a seguir compila o programa Prog.cpp e pede para linkar em modo estático (-static) a biblioteca nomeLib que esta localizada em pathLib.

Exemplo:

```
g++ Prog.cpp -static -LpathLib -lnomeLib
```

46.5 Montando uma biblioteca dinâmica (libNome.so)

Roteiro para criar uma biblioteca dinâmica no Linux:

1. Cria o arquivo de cabeçalho *.h (declaração das funções e classes)

```
emacs TPonto.h
class TPonto { .....};
```

2. Cria o arquivo de código *.cpp (definição das funções)

```
emacs TPonto.cpp
/*Define funcoes da classe*/
```

3. Compila os arquivos de código (*.cpp) gerando os arquivos *.o

A opção -fPIC, gera o código com posicionamento independente, podendo o mesmo ser carregado em qualquer endereço.

```
g++ -fPIC -c TPonto.cpp -o TPonto.o
```

4. Cria a biblioteca dinâmica

A opção -WL passa informações para o linker ld.

```
-g++ -shared -Wl,-soname,TPonto.so.1 -o libTPonto.so.1.0 TPonto.o
```

5. Copia a lib para /usr/local/lib (como root)

```
cp libTPonto.so.1.0 /usr/local/lib
```

6. Pode-se criar links simbólicos para a lib

```
cd /usr/local/lib/
ln -s libTPonto.so.1.0 libTPonto.so.1
ln -s libTPonto.so.1.0 libTPonto.so
```

7. Publica a lib

#inclue na tabela de bibliotecas dinâmicas, cria link, e inclue em /etc/ld.so.cache

```
/sbin/ldconfig
```

46.5.1 Vantagens/desvantagens da biblioteca dinâmica

- Em uma biblioteca estática o programa é grande porque inclui todas as bibliotecas.
- Quando o programa é linkado com bibliotecas dinâmicas, o mesmo fica menor, pois as bibliotecas são carregadas em tempo de execução.
- O programa que usa a biblioteca dinâmica não precisa ser recompilado.

Veja a seguir um arquivo makefile para gerar o programa e87-Polimorfismo.cpp, anteriormente apresentado.

Listing 46.4: Arquivo makefile com bibliotecas estáticas e dinâmicas.

```
##### DEFINICOES#####
ARQUIVOS= e87-TCirculo.cpp e87-TElipse.cpp e87-TPonto.cpp e87-Polimorfismo.
  cpp
OBJETOS= e87-TCirculo.o e87-TElipse.o e87-TPonto.o
DIRINCLUDE = -I/usr/include/g++ -I/usr/include
DIRLIBD = /home/andre/Andre/ApostilasPessoais/ApostilaProgramacao/Exemplos/
  cursocpp/Cap-GNU/biblioteca
PARAMETROSLIB= -lm
COMPILADOR = g++
LIBS = TPonto
LIBD = TPonto
PROG = e87-Polimorfismo.cpp

#-----Lista de opções.
#list:
# echo "obj:      Gera objetos comuns\
# exe:          Gera executável comum"
# all:         obj exe \
# libs:        Gera biblioteca estática \
# exes:        Gera executável usando biblioteca estática \
# alls:        libs exelibs \
# libd:        Gera biblioteca dinâmica \
# exed:        Gera executável usando biblioteca dinâmica \
# libs_libtool: Gera biblioteca estatica usando lib_tool \
# exes_libtool: Gera executável usando biblioteca estatica e libtool \
# init_doc:    Inicializa o doxygen \
# doc:         Gera documentação a partir de código documentado \
# clean:       Apaga arquivos "

#-----Compilação padrão
all : obj exe

obj : $(ARQUIVOS)
      $(COMPILADOR) -c $(ARQUIVOS) $(DIRINCLUDE) $(DIRECL)

exe : $(PROG) $(OBJETOS)
      $(COMPILADOR) $(PROG) $(OBJETOS) $(DIRINCLUDE) $(DIRETORIOLIB) $(
        PARAMETROSLIB) -o e87-Polimorfismo

#-----Criando biblioteca estática
alls : libs exelibs
```



```

libs : $(OBJETOS)
        ar -q libTPonto.a $(OBJETOS)
#ar -cru libTPonto.a $(OBJETOS)
        ranlib libTPonto.a          #publica a lib

#Criando executavel usando a biblioteca estática
exelibs : libs
        $(COMPILADOR) e87-PolimorfismoStatic.cpp -static -L. -l$(LIBS) -o e87-
        PolimorfismoStatic

#-----Criando biblioteca dinâmica
alld : objd libd

objd : $(ARQUIVOS)
        $(COMPILADOR) -fPIC -c $(ARQUIVOS) $(DIRINCLUDE) $(DIRECL)

libd : $(OBJETOS)
        $(COMPILADOR) -shared -Wl,-soname,TPonto.so.1 -o libTPonto.so.1.0 $(
        OBJETOS)
        ln -s libTPonto.so.1.0 libTPonto.so.1
        ln -s libTPonto.so.1.0 libTPonto.so
        /sbin/ldconfig          #publica a biblioteca (como root)

#-----Criando executavel usando a biblioteca dinâmica
exelibd : e87-PolimorfismoDinamic.cpp libd
        $(COMPILADOR) e87-PolimorfismoDinamic.cpp -L$(DIRLIBD) -l$(LIBD) -o e87-
        PolimorfismoDinamic

#-----Limpeza.
clean :
        rm -f *.o *.obj *.so* a.out e87-PolimorfismoDinamic e87-
        PolimorfismoStatic *.*~ *~ libTPonto.*

#=====USANDO LIBTOOL=====
#Para ficar portátil, todas as etapas devem ser realizadas com o libtool
#obj_libtool : $(ARQUIVOS)
#        libtool $(COMPILADOR) -c $(ARQUIVOS)

#-----Criando biblioteca estatica usando o libtool
libs_libtool: $(ARQUIVOS)
        libtool $(COMPILADOR) -o libTPonto.a $(ARQUIVOS)

#-----Criando executavel usando libtool com biblioteca estatica
#Incompleto
exes_libtool: e87-PolimorfismoStatic.cpp libs_libtool
        libtool $(COMPILADOR) e87-PolimorfismoStatic.cpp -static -L. -l$(LIBS)
        -o e87-PolimorfismoStatic-libtool

#-----Criando biblioteca dinâmica usando o libtool
libd_libtool: $(ARQUIVOS)
        libtool $(COMPILADOR) -o libTPonto.la $(ARQUIVOS)

#-----Criando executavel usando libtool com biblioteca dinâmica
#Incompleto

```

```

exed_libtool: e87-PolimorfismoStatic.cpp libd_libtool
             libtool $(COMPILADOR) e87-PolimorfismoDinamic.cpp libTPonto.la -o e87
             -PolimorfismoDinamic-libtool

```

```

#-----Criando documentação com doxygen

```

```

init_doc:
    doxygen -g doxygen.config

```

```

doc : $(ARQUIVOS)
    doxygen doxygen.config

```

Faça cópias do e87-Polimorfismo.cpp criando os arquivos e87-PolimorfismoStatic.cpp e e87-PolimorfismoDinamic.cpp e então execute a sequência ilustrada a seguir.

Listing 46.5: Arquivo mostrando o uso do makefile.

```

[root@mercurio biblioteca]# make clean
rm -f *.o *.obj *.so* a.out e87-PolimorfismoDinamic e87-PolimorfismoStatic
    *.*~ *~ libTPonto.*

[root@mercurio biblioteca]# make all
g++ -c e87-TCirculo.cpp e87-TElipse.cpp e87-TPonto.cpp e87-Polimorfismo.cpp
    -I/usr/include/g++ -I/usr/include
g++ e87-Polimorfismo.cpp e87-TCirculo.o e87-TElipse.o e87-TPonto.o -I/usr/
    include/g++ -I/usr/include -lm -o e87-Polimorfismo

[root@mercurio biblioteca]# make alls
ar -q libTPonto.a e87-TCirculo.o e87-TElipse.o e87-TPonto.o
ranlib libTPonto.a #publica a lib
g++ e87-PolimorfismoStatic.cpp -static -L. -lTPonto -o e87-PolimorfismoStatic

[root@mercurio biblioteca]# make alld
g++ -fPIC -c e87-TCirculo.cpp e87-TElipse.cpp e87-TPonto.cpp e87-Polimorfismo
    .cpp -I/usr/include/g++ -I/usr/include
g++ -shared -Wl,-soname,TPonto.so.1 -o libTPonto.so.1.0 e87-TCirculo.o e87-
    TElipse.o e87-TPonto.o
ln -s libTPonto.so.1.0 libTPonto.so.1
ln -s libTPonto.so.1.0 libTPonto.so
/sbin/ldconfig #publica a biblioteca (como root)

[root@mercurio biblioteca]# make libs_libtool
libtool g++ -o libTPonto.a e87-TCirculo.cpp e87-TElipse.cpp e87-TPonto.cpp
    e87-Polimorfismo.cpp
ar cru libTPonto.a
ranlib libTPonto.a

[root@mercurio biblioteca]# make libd_libtool
libtool g++ -o libTPonto.la e87-TCirculo.cpp e87-TElipse.cpp e87-TPonto.cpp
    e87-Polimorfismo.cpp
rm -fr .libs/libTPonto.la .libs/libTPonto.* .libs/libTPonto.*
ar cru .libs/libTPonto.al
ranlib .libs/libTPonto.al
creating libTPonto.la
(cd .libs && rm -f libTPonto.la && ln -s ../libTPonto.la libTPonto.la)

```

46.5.2 Usando uma biblioteca dinâmica

```
//Inclue o arquivo de inclusão de bibliotecas dinâmicas
#include <dlfcn.h>
#include <fstream.h>...
main()
{
//Cria ponteiro para a lib
void* ptrLib;
//Cria ponteiro para função da lib
void (*ptrFuncaoLib)();
//Carrega a lib
//dlopen(const char* fileName, int flag);
ptrLib = dlopen("nomeLib.so.1.0",RTLD_LAZY);
//Verifica se não tem erro com a função dlerror
//const char* dlerror();
cout << dlerror();
//Obtém endereço da função
//void* dlsym(void* handle,char* simbolo);
ptrFuncaoLib = dlsym(ptrLib,"NomeFuncaoNaLib");
//Usa a função
int x = (*ptrFuncaoLib)();
//Fecha a lib
//int dlclose(void * handle);
dlclose(ptrLib);
}
```

46.6 Sentenças para bibliotecas

- O padrão para o nome da lib é: libNome.so.versao.subversao.release.
- Você pode acrescentar novas paths para bibliotecas dinâmicas modificando a variável de ambiente LD_LIBRARY_PATH.

Capítulo 47

Libtool

47.1 Introdução ao libtool

Como será descrito nos capítulos seguintes, o libtool é mais um programa da gnu, que facilita o desenvolvimento de bibliotecas multiplataforma. O mesmo é usado para desenvolvimento de bibliotecas no ambiente Linux.

Vantagens do uso do libtool

- Maior elegância
- Integrado ao autoconf e automake
- Maior portabilidade
- Trabalha com bibliotecas estáticas e dinâmicas

47.2 Forma de uso do libtool

Como faço para usar o libtool ?

De um modo geral, basta digitar o comando **libtool** seguido do comando que você usaria para compilar seu programa ou biblioteca.

Exemplos de uso do libtool estão listados no diretório do libtool.

Veja a seguir a saída do comando **libtool - - help**.

Listing 47.1: Arquivo libtool -help.

```
Usage: libtool [OPTION]... [MODE-ARG]...
```

```
Provide generalized library-building support services.
```

```
  --config          show all configuration variables
  --debug          enable verbose shell tracing
-n, --dry-run      display commands without modifying any files
  --features       display basic configuration information and exit
  --finish         same as '--mode=finish'
  --help          display this help message and exit
  --mode=MODE      use operation mode MODE [default=inferred from MODE-ARGS]
```

```
--quiet          same as '--silent'
--silent         don't print informational messages
--version        print version information
```

MODE must be one of the following:

```
clean           remove files from the build directory
compile         compile a source file into a libtool object
execute         automatically set library path, then run a program
finish         complete the installation of libtool libraries
install         install libraries or executables
link           create a library or an executable
uninstall       remove libraries from an installed directory
```

MODE-ARGS vary depending on the MODE. Try 'libtool --help --mode=MODE' for a more detailed description of MODE.

47.3 Criando uma biblioteca sem o libtool

Reveja a seguir como criar uma biblioteca estática sem uso do libtool.

Exemplo:

```
ar cru libNome.a a.o b.o c.o
ranlib libNome.a
```

Para criar uma biblioteca estática usando um arquivo makefile, anteriormente apresentado execute o comando:

Exemplo:

```
make clean
make libs
```

A saída gerada pelo makefile é dada por:

```
[andre@mercurio libtool-biblioteca]$ make libs
g++ -c -o e87-TCirculo.o e87-TCirculo.cpp
g++ -c -o e87-TElipse.o e87-TElipse.cpp
g++ -c -o e87-TPonto.o e87-TPonto.cpp
ar -q libTPonto.a e87-TCirculo.o e87-TElipse.o e87-TPonto.o
ranlib libTPonto.a
```

47.4 Criando uma biblioteca estática com o libtool

Agora, o mesmo exemplo usando o libtool.

Exemplo:

```
libtool g++ -o libTPonto.a e87-TPonto.cpp
                    e87-TElipse.cpp e87-TCirculo.cpp
```

Veja a seguir a saída gerada pelo libtool.

```
mkdir .libs
ar cru libTPonto.a
ranlib libTPonto.a
```

47.5 Criando uma biblioteca dinâmica com o libtool

Agora, o mesmo exemplo usando o libtool e biblioteca dinâmica. Observe que a única alteração é o nome da biblioteca, que agora se chama libTPonto.la.

```
Exemplo:
libtool g++ -o libTPonto.la e87-TPonto.cpp
e87-TElipse.cpp e87-TCirculo.cpp
```

Veja a seguir a saída gerada pelo libtool.

```
rm -fr .libs/libTPonto.la .libs/libTPonto.* .libs/libTPonto.*
ar cru .libs/libTPonto.al
ranlib .libs/libTPonto.al
creating libTPonto.la (cd .libs && rm -f libTPonto.la
&& ln -s ../libTPonto.la libTPonto.la)
```

47.6 Linkando executáveis

Formato usual:

```
Exemplo:
g++ -o nomeExecutável nomePrograma.cpp libNome.la
```

Formato usando o libtool:

```
Exemplo:
libtool g++ -o nomeExecutável nomePrograma.cpp libNome.la
```

47.7 Instalando a biblioteca

Formato usual:

```
Exemplo:
//como root
cp libNome.a /usr/lib
ranlib /usr/lib/libNome.a
```

Formato usando o libtool:

Exemplo:

```
libtool cp libNome.a /usr/lib/
//ou
libtool install -c libNome.a /usr/lib/libNome.la
//ou
libtool install -c .libs/libNome.a /usr/lib/libNome.so.0.0
libtool install -c .libs/libNome.a /usr/lib/libNome.la
libtool install -c .libs/libNome.a /usr/lib/libNome.a
```

47.8 Modos do libtool

Para saber mais sobre o funcionamento de cada um dos módulos abaixo listados, execute o comando: **libtool -help -mode = MODO**.

Compilação Atua chamando o compilador do sistema:

```
libtool -help -mode=compile
```

Linkagem Atua executando a linkagem:

```
libtool -help -mode=link
```

Instalação Atua instalando o programa:

```
libtool -help -mode=install
```

Execução Atua executando o programa:

```
libtool -help -mode=execute
```

Desinstalação Atua desinstalando o programa:

```
libtool -help -mode=uninstall
```

47.9 Sentenças para o libtool

- Para executar o gdb com o libtool use:

```
libtool gdb nomePrograma
```

- Para obter informações do libtool:

```
libtool - -help
```

- Para gerar apenas bibliotecas estáticas, passar o flag:

```
--disable-shared
```

- Durante o desenvolvimento costuma-se usar biblioteca estática com opção de debugagem.
- Em 9/2001 o libtool ainda não era totalmente compatível com C++.

-
- Leia o livro “GNU AUTOCONF, AUTOMAKE, AND LIBTOOL” disponível gratuitamente no site (<http://sources.redhat.com/autobook/>).
 - Bibliotecas compartilhadas usam a especificação PIC (position independent code).

Capítulo 48

Debug (Depuradores, Debuggers)

Bem, você é um bom programador, mas..., ainda existem alguns probleminhas, e você terá de rastrear o seu código para eliminar aqueles pequenos insetos.

Não adianta, você vai ter de debugar seu código.

O GNU Linux/Unix tem o gdb um debug em modo texto, e seus frontends xgdb e kdbg.

Antes de mais nada, para poder debugar o seu código, você precisa acrescentar as informações para o debug passando a opção de compilação -g (CPPFLAGS= -g).

Desta forma o gdb poderá examinar o seu executável (ou o arquivo core) para verificar o que aconteceu.

48.1 Comandos do gdb

Apresenta-se na Tabela 48.1 uma lista com os comandos do gdb.

48.2 Exemplo de uso do gdb

Um pequeno exemplo de uso do gdb.

```
Exemplo
(gdb) Run           //Roda o programa
(gdb) backtrace     //Mostra a pilha (o último comando executado)
(gdb) break 23      //Acrescenta breakpoint na linha 23
(gdb) list          //Mostra o código fonte perto do breakpoint
(gdb) p var         //Mostra o conteúdo da variável
(gdb) c            //Continua execução
```

48.3 Sentenças para o gdb

- No Linux, quando um programa trava ou é encerrado de forma inesperada, é gerado um arquivo core. O arquivo core pode ser aberto pelo gdb para localizar a posição onde o programa travou.

Tabela 48.1: Comandos do gdb.

Comando	Ação
<code>gdb</code>	Executa o debugger.
<code>run prog</code>	Executa o programa <code>prog</code> .
<code>run prog arg</code>	Roda o programa com os argumentos.
<code>bt</code>	Apresenta um rastreamento da pilha.
<code>break func</code>	Cria breakpoint na função <code>func</code> .
<code>list arq.cpp</code>	Visualiza o <code>arq.cpp</code> .
<code>break 25 (ou b25)</code>	Acrescenta breakpoint na linha 25.
<code>delete (d)</code>	Deleta os breakpoint.
<code>c</code>	Continua.
<code>step</code>	Executa um passo.
<code>step 10</code>	Executa os próximos 10 passos.
<code>next</code>	Executa uma linha.
<code>next 12</code>	Executa as próximas 12 linhas.
<code>print var</code>	Imprime o conteúdo da variável.
<code>what atributo</code>	Mostra conteúdo do atributo/variável.
<code>quit</code>	Abandona o debug.
<code>help com</code>	Help sobre o comando.

- No gnu você pode simplificar o trabalho de debugagem incluindo a macro `__FUNCTION__` que imprime o nome da função que esta sendo executada.

Exemplo:

```
cout << " na função : " << __FUNCTION__ << endl;
```

Capítulo 49

Profiler (gprof)

Um profiler é um programa utilizado para avaliar o desempenho do seu programa, permitindo encontrar os gargalos (pontos onde o programa demora mais).

O profiler apresenta um gráfico com o tempo de execução de cada função.

Exemplo:

```
//Compila incluindo opção -pg
g++ -pg -c ex-funcaoobjeto1.cpp
//cria o executável a.out que é aberto pelo gprof
//executa o gprof
gprof --brief -p
//saída do gprof
[andre@mercurio Cap4-STL]$ gprof --brief -p
Flat profile:
Each sample counts as 0.01 seconds. no time accumulated
% cumulative self self total time seconds seconds calls Ts/call Ts/call name
0.00 0.00 0.00 28 0.00 0.00 _Deque_iterator<int, int &, int *, 0>::_S_buffer_size(void)
0.00 0.00 0.00 21 0.00 0.00 _Deque_iterator<int, int &, int *, 0>::operator!=
(_Deque_iterator<int, int &, int *, 0> const &) const
0.00 0.00 0.00 17 0.00 0.00 _Deque_iterator<int, int &, int *, 0>::operator-
(_Deque_iterator<int, int &, int *, 0> const &) const
0.00 0.00 0.00 15 0.00 0.00 _Deque_base<int, allocator<int>, 0>::~~_Deque_base(void)
0.00 0.00 0.00 14 0.00 0.00 void destroy<int *>(int *, int *) 0.
```

Observe a direita o nome da função e a esquerda o tempo de execução.

49.1 Sentenças para o profiler:

- Para aprender a usar o gprof, baixe e leia o manual do gprof do site da gnu.
- Se você quer um compilador mais rápido e usa máquinas com processadores pentium pense em usar o pgcc. Um compilador descentente do compilador da gnu e otimizado para processadores pentium. Procure por pgcc na internet.
- Você só deve se preocupar com performance (e com o gprof) depois que for um bom programador. Primeira faça com que os programas funcionem, a seguir se preocupe com bugs, faça a documentação,....., depois de tudo se preocupe com a performance.

Capítulo 50

Versão de Depuração, Final e de Distribuição

Neste capítulo apresenta-se as opções para criar a versão de depuração e a versão final de seu programa. Apresenta-se ainda as formas de distribuição de programas.

50.1 Versão debug, release e de distribuição

A medida que o programa é desenvolvido e os bugs corrigidos, costuma-se trabalhar com uma versão de debugagem (passando a opção -g para o compilador). Depois, quando deseja-se distribuir um release do programa, eliminam-se todas as opções de debugagem e colocam-se opções de otimização.

50.1.1 Versão debug

1. Ativar a opção de debugagem (-g).
2. Ativar todos os warnigs (-Wall).

50.1.2 Versão final (release)

1. Desativar todas as opções de debugagem (tirar -g).
2. Ativar as opções de otimização (-O1,-O2,-O3).
3. Ativar todos os warnings (-Wall).

50.1.3 Distribuição dos programas e bibliotecas

Uma biblioteca pode ser vendida, distribuindo-se os arquivos de cabeçalho (*.h) e os arquivos da biblioteca (*.lib).

Um programador que comprou as bibliotecas, pode usar as funções e objetos da biblioteca consultando os manuais e os arquivos de cabeçalho (*.h).

Observe que como os arquivos *.cpp não são distribuídos, o programador não tem acesso a forma como as funções foram implementadas, isto é, não tem como avaliar a qualidade da

biblioteca. Este é o formato de distribuição de bibliotecas no ambiente Windows. Um sistema proprietário que esconde de quem compra o programa o seu código, o seu real funcionamento (suas qualidades e seus problemas).

Um formato mais moderno e democrático é distribuir tanto os arquivos de cabeçalho (*.h) como os de implementação (*.cpp), este é o sistema de distribuição do Linux.

Para distribuir seu código você pode utilizar uma das opções descritas a seguir.

Distribuir o seu código fonte em um arquivo .tar.gz

1. Gera o arquivo de distribuição

Ex: `tar -cvzf nomeArquivo.tar.gz path_do_programa`

2. Permite o acesso dos usuários pela internet ou pela distribuição de disketes (zip, cd).

Distribuir o seu código fonte com patches (atualizações)

Além de distribuir o seu código com o arquivo .tar.gz você pode distribuir upgrades, isto é, distribuições que acrescentam apenas as modificações que foram realizadas no código. A grande vantagem é que o usuário precisa baixar arquivos pequenos. O roteiro abaixo mostra como distribuir atualizações de código com patches.

Programador_etapa_1:

1. Gera o programa

```
make
testa se o programa esta ok...
```

2. Gera a distribuição .tar.gz

```
//se o arquivo makefile foi configurado para gerar uma distribuição
make dist
//ou cria um arquivo compactando todo o código com o tar
tar -cvzf prog.tar.gz path_do_programa
```

Usuário_etapa_1:

1. Baixa, descompacta, compila e instala o programa.

```
ftp site_com_o_programa
login
get prog.tar.gz
tar -xvzf prog.tar.gz
./configure
make
make install
```


Programador_etapa_2:

1. Faz atualizações no código (upgrades)

Edita os arquivos com o programa e inclue atualizações e correções de bugs

2. Gera os arquivos com as diferenças (coisas novas nos códigos)

```
diff arq1.cpp arq2.cpp > arq1.dif
```

3. Gera o pacote com as atualizações

```
tar -cvzf path-versaold-versaonova.tar.gz *.dif
```

Usuário_etapa_2:

1. Baixa e descompacta o arquivo path-versaold-versaonova.tar.gz.

```
ftp site_com_o_programa
login
get path-versaold-versaonova.tar.gz
tar -xvzf path-versaold-versaonova.tar.gz
```

2. Executa o programa patch para que o código antigo (arq.1) seja alterado, incluindo as alterações da nova versão, ou seja, gera um novo arquivo arq1 somando ao arquivo arq1 as novidades listadas no arquivo arq1.dif.

```
cp arq1.cpp arq1.cpp~
patch arq1.cpp arq1.dif
```

3. Configura, compila e instala o programa

```
./configure
make
make install
```

4. As alterações realizadas no arquivo arq.1 podem ser revertidas (voltar a ser o que era) executando-se

```
patch -r arq1.cpp arq1.dif
```

Distribuir o seu código fonte através do uso do CVS

Você terá de ter um servidor de CVS instalado e funcionando. Deve criar um repositório para o seu programa (com releases,...) e um sistema que permita aos usuários baixar os arquivos no servidor CVS para serem compilados.

O uso do CVS é descrito no Capítulo 54.

Exemplo:

```
//baixando para seu micro um programa utilizando o cvs
```

```
....
```

Distribuir o programa (e os fontes) em pacotes rpm

O desenvolvimento de arquivo rpm é um pouco complexo e esta fora do escopo desta apostila. Dê uma olhada no kdevelop e verifique se o mesmo já esta gerando distribuições no formato rpm.

50.2 Sentenças para distribuição de código fonte

- Se você esta programando no Linux é bem provável que já tenha baixado e compilado o kernel do Linux. O mesmo é disponibilizado no site <http://kernel.org>. Vá ao site do kernel do Linux e compare o tamanho das atualizações do kernel, isto é, compare os arquivos `kernel-versao.tar.bz2` e `patch-versao.bz2`. Veja o resultado aí em baixo, de 23576Kb para 807Kb.

```
patch-2.4.18.bz2 807 KB 25-02-2002 19:44:00
```

```
linux-2.4.18.tar.bz2 23596 KB 25-02-2002 19:40:00 File
```

- Use o `patch`, a atualização dos programas fica muito mais rápida pois os arquivos de atualizações são pequenos e podem ser baixados com extrema rapidez na internet.

Capítulo 51

Documentação de Programas Usando Ferramentas Linux

Apresenta-se neste capítulo o uso do formato `JAVA_DOC` para embutir documentação em seus programas. O uso do programa `doxygen` para gerar a documentação em diferentes formatos (`html/tex/rtf`). Cita-se ainda os formatos `sgml` e `xml` para geração de manuais profissionais.

51.1 Introdução a documentação de programas

A documentação é uma etapa fundamental para compreensão, aperfeiçoamento e manutenção de programas.

Existem atualmente alguns programas e padrões para documentação de códigos em `C++`. Vou descrever brevemente o formato `JAVA_DOC` que é aceito pelo gerador de documentação `DOXYGEN`.

Como funciona ?

Você inclui em seu código um conjunto de tags. Estes tags não interferem na compilação de seu programa, apenas incluem informações que serão identificadas por um programa externo (o `doxygen`), para gerar a documentação das classes, dos atributos e dos métodos.

- A primeira etapa é incluir a documentação nos arquivos `*.h` e `*.cpp`, veja seção 51.2.
- A segunda etapa consiste em executar o `doxygen` (que já foi instalado e configurado) para gerar a documentação, veja seção 51.3.

51.2 Documentação embutida no código com `JAVA_DOC`

O formato `JAVA_DOC` é amplamente utilizado para documentação de programas em `C++`, também é válido para documentação de códigos em `JAVA` e `IDL`.

51.2.1 Exemplo de código documentado

Para inserir um breve comentário utilize três barras invertidas.

```
///Breve comentário (apenas uma linha).
```

Para inserir um breve comentário e um comentário mais detalhado use

```
/** Breve comentário
 * Comentário mais detalhado
 *.....
 *@class TNome
 *@file NomeArquivo
 */
class TNome
{
.
.
```

No exemplo acima observe a posição do breve comentário e do comentário detalhado. Veja ainda a forma de definição do nome da classe e do arquivo.

Para inserir um comentário embutido use:

```
int a; /**< Comentário pequeno, embutido*/
```

Observe que o comentário inicia com um `/**<` e termina com um `*/`. Veja a seguir outro exemplo.

```
/** Um enumerador (breve descrição)
 * Descrição detalhada do enumerador
 */
enum ENome {
segunda, /**< Comentário pequeno, embutido*/
terca,/**< Comentário pequeno, embutido*/
}
/** Breve descrição da função
 * Descrição detalhada da função
 *@param int a
 *@param int b
 *@return retorna a soma (int)
 *@see
 */
int Soma(int a, int b);
};
```

No exemplo acima passa-se o nome dos parâmetros de retorno.

51.2.2 Sentenças para documentação java_doc

- Se houver documentação duplicada: na classe `[*.h]` e nos arquivos de definição `[*.cpp]`. Vai usar a breve descrição incluída na classe `[*.h]` e a documentação detalhada incluída na definição da função `[*.cpp]`.
- Aceita os tags: `class`, `struct`, `union`, `enum`, `fn`(função), `var` (atributos), `def` (define), `file`(arquivo), `namespace` (um namespace), `brief` (breve comentário).
- Para inserir uma lista:

```
/**
```

```

*           Nome Lista
           -Ítem      A
                           -#      SubÍtem A.1
                           -#      SubÍtem A.2
           -Ítem      B
                           -#      SubÍtem B.1
                           -#      SubÍtem B.2

*/

```

- Para criar grupos:

```

/** @addgroup <NomeGrupo> */
/** @ingroup NomeGrupo */
/** @defgroup NomeGrupo */

```

- Para detalhes, veja a documentação do formato JAVA_DOC.

51.3 Tutorial de configuração e uso do DOXYGEN

O doxygen é um programa que gera a documentação (API) a partir de informações incluídas no código. As informações são incluídas no código utilizando-se o formato JAVA_DOC (veja seção 51.2).

Breve tutorial de uso do doxygen

1. Baixar o programa doxygen (ou usar os CDs da sua distribuição LINUX).
Atualmente a maioria das distribuições Linux inclui o doxygen.
Você pode obter o doxygen no site <http://www.stack.nl/~dimitri/doxygen/>.
2. Instalar o doxygen usando o arquivo .tar.gz (./configure && make && make install), ou usando pacote rpm.
3. Criar um arquivo de configuração do projeto a ser documentado.
doxygen -g nomeArquivoConfiguracao
4. Incluir a documentação em seu código (veja seção 51.2 na página 451).
5. Executar o doxygen (gera por default saída html).
doxygen nomeArquivoConfiguracao
6. Para gerar saída latex setar a variável
GENERATE_LATEX = YES
Vá para o diretório com os arquivos do latex e execute
(make && make ps && make pdf).
Para gerar 2 folhas por página, vá para o diretório
com os arquivos do latex e execute:
(make && make ps_2on1 && make pdf_2on1).
7. Você pode adicionar ao doxygen o programa da graphviz. O mesmo é usado para gerar diagramas de relacionamento das diversas classes. Procure pelo programa no site (<http://www.research.att.com/sw/tools/graphviz/>).

Apresenta-se nas linhas abaixo um arquivo de configuração do doxygen. O arquivo inclui comentários dentro de parenteses.

```

..# Doxyfile 0.1
#-----
# General configuration options
#-----
PROJECT_NAME       = "Biblioteca de objetos - LIB_LMPT" (nome projeto)
PROJECT_NUMBER    = 0.4 (versão)
OUTPUT_DIRECTORY  = /home/andre/Andre/Desenvolvimento/LIB_LMPT-api/ (diretório de saída)
OUTPUT_LANGUAGE   = English (linguagem)
EXTRACT_ALL       = YES (extrair todas as informações)
EXTRACT_PRIVATE   = YES (incluir atributos/funções privados)
EXTRACT_STATIC    = YES (incluir atributos/funções estáticas)
HIDE_UNDOC_MEMBERS = NO
HIDE_UNDOC_CLASSES = NO
#-----
# configuration options related to the LATEX output
#-----
GENERATE_LATEX     = YES
LATEX_OUTPUT      = (diretório opcional, por default cria diretório latex)
COMPACT_LATEX     = NO
PAPER_TYPE        = a4wide (formato da folha)
EXTRA_PACKAGES    =
LATEX_HEADER      = (link para header)
PDF_HYPERLINKS    = YES (gerar links para pdf)
USE_PDFLATEX      = YES (gerar arquivo pdf)
LATEX_BATCHMODE   = NO
#-----
# configuration options related to the RTF output
#-----
GENERATE_RTF      = NO
RTF_OUTPUT       =
COMPACT_RTF      = NO
RTF_HYPERLINKS  = NO
RTF_STYLESHEET_FILE =
RTF_EXTENSIONS_FILE =
#-----
# configuration options related to the man page output
#-----
GENERATE_MAN      = NO
MAN_OUTPUT       =
MAN_EXTENSION    =
MAN_LINKS        = NO
#-----
# configuration options related to the XML output
#-----
GENERATE_XML      = NO
#-----
# Configuration options related to the preprocessor
#-----
ENABLE_PREPROCESSING = YES
MACRO_EXPANSION     = NO
EXPAND_ONLY_PREDEF  = NO
SEARCH_INCLUDES     = YES
INCLUDE_PATH        =
INCLUDE_FILE_PATTERNS =
PREDEFINED          =
EXPAND_AS_DEFINED   =
#-----
# Configuration::addtions related to external references
#-----
TAGFILES          =
GENERATE_TAGFILE  =
ALLEXTERNALS     = NO
PERL_PATH         =
#-----
# Configuration options related to the dot tool
#-----
HAVE_DOT         = YES
CLASS_GRAPH      = NO (se true, desabilita geração do grafico das heranças e gera gráfico das heranças e relações)
COLLABORATION_GRAPH = YES (grafico da hierarquia e relacionamentos)
INCLUDE_GRAPH    = YES (grafico dos arquivos include)
INCLUDED_BY_GRAPH = YES
GRAPHICAL_HIERARCHY = YES (gera diagrama de heranças)
DOT_PATH         = /home/SYSTEM/gv1.7c/bin
MAX_DOT_GRAPH_WIDTH  = 1024
MAX_DOT_GRAPH_HEIGHT = 1024
GENERATE_LEGEND   = YES
DOT_CLEANUP      = YES (deleta arquivos temporários)
#-----
# Configuration::addtions related to the search engine
#-----
SEARCHENGINE     = NO
CGI_NAME         =
CGI_URL          =
DOC_URL          =
DOC_ABSPATH     =
BIN_ABSPATH     =
EXT_DOC_PATHS   =

```

Observe na listagem acima a opção de uso do programa auxiliar “dot tool”. O mesmo é utilizado para gerar diagramas de relacionamento entre as classes.

Para obter informações gerais sobre o doxygen execute:

```
doxygen --help
```

A saída do comando (doxygen - -help) é dada por

Listing 51.1: Saída do comando doxygen -help.

```
Doxygen version 1.2.8.1 Copyright Dimitri van Heesch 1997-2001

You can use doxygen in a number of ways:

1. Use doxygen to generate a template configuration file:
doxygen [-s] -g [configName]
If - is used for configName doxygen will write to standard output.

2. Use doxygen to update an old configuration file:
doxygen [-s] -u [configName]

3. Use doxygen to generate documentation using an existing configuration file:
doxygen [configName]
If - is used for configName doxygen will read from standard input.

4. Use doxygen to generate a template style sheet file for RTF, HTML or Latex.
RTF:
doxygen -w rtf styleSheetFile
HTML:
doxygen -w html headerFile footerFile styleSheetFile [configFile]
LaTeX:
doxygen -w latex headerFile styleSheetFile [configFile]

5. Use doxygen to generate an rtf extensions file RTF:
doxygen -e rtf extensionsFile

If -s is specified the comments in the config file will be omitted. If
configName is omitted 'Doxyfile' will be used as a default.
```

51.4 Exemplo de programa documentado

A listagem a seguir apresenta um exemplo de programa documentado.

Listing 51.2: Exemplo de código documentado no formato JAVA_DOC para uso com o programa doxygen.

```
#ifndef TTeste_h
#define TTeste_h

/*
=====
PROJETO:      Biblioteca LIB_LMPT
              Assunto/Ramo: TTeste...
=====
Desenvolvido por:
```

```

        Laboratorio de Meios Porosos e Propriedades Termofisicas
        [LMPT].
@author   André Duarte Bueno
@file    TTeste.h
@begin   Sat Sep 16 2000
@copyright (C) 2000 by André Duarte Bueno
@email   andre@lmpt.ufsc.br
*/

//-----
//Bibliotecas C/C++
//-----

//-----
//Bibliotecas LIB_LMPT
//-----
//#include <Base/_LIB_LMPT_CLASS.h>

/*
=====
Documentacao CLASSE: TTeste
=====
*/
/**
@short
    Classe de teste das diversas classes da LIB_LMPT.
    O objetivo é dentro da main criar e chamar TTeste
    que cria e chama as demais classes.

Assunto:      Teste da LIB_LMPT
Superclasse:  TTeste
@author       André Duarte Bueno

@version      versão...
@see          veja assunto...
*/
class TTeste
{
//-----Atributos
private:
protected:
public:

//-----Construtor
    //Construtor
    TTeste ()
    {
    };

//-----Destrutor
    //Destrutor
    virtual ~ TTeste ()
    {
    };

```



```
//-----Métodos
private:
protected:
public:

    /** Função principal, executada por main.
        Vai solicitar ao usuário o tipo de objeto a ser criado,
        criar o objeto e passar o controle
        do programa para o objeto criado */
    void Run ();
};

//-----Friend
//Declaração de Funções Friend
//ostream& operator<< (ostream& os, TTeste& obj);
//istream& operator>> (istream& is, TTeste& obj);
#endif
```

51.5 Exemplo de diagramas gerados pelo doxygen

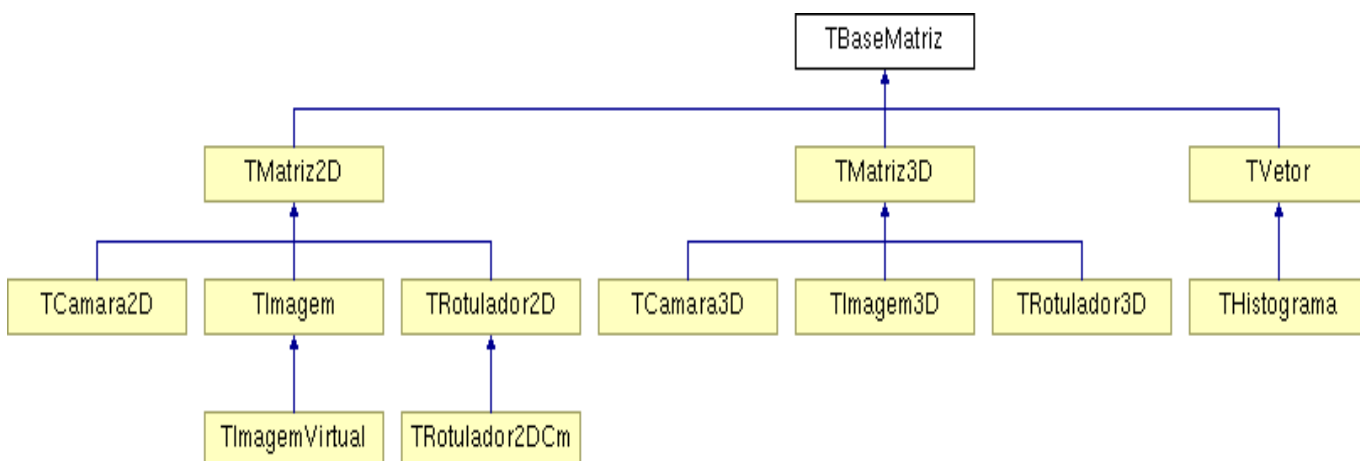
Você pode instalar em seu sistema o programa da graphvis. O programa da graphvis é utilizado para gerar diagramas das classes, ilustrando as diversas classes e seus relacionamentos. Gera ainda diagramas das dependências dos arquivos. Procure pelo programa no site (<http://www.research.att.com/sw/tools/graphviz/>).

Você pode configurar o doxygen para que use o programa da graphvis, possibilitando assim a inclusão dentro da documentação (html, tex), de Figuras ilustrando as hierarquias das diversas classes.

Para ilustrar a documentação api gerada pelo doxygen, incluí na distribuição desta apostila o arquivo LIB_LMPT-api.tar.gz. Você pode descompactar este arquivo e ver como fica a documentação gerada com o seu browser (netscape).

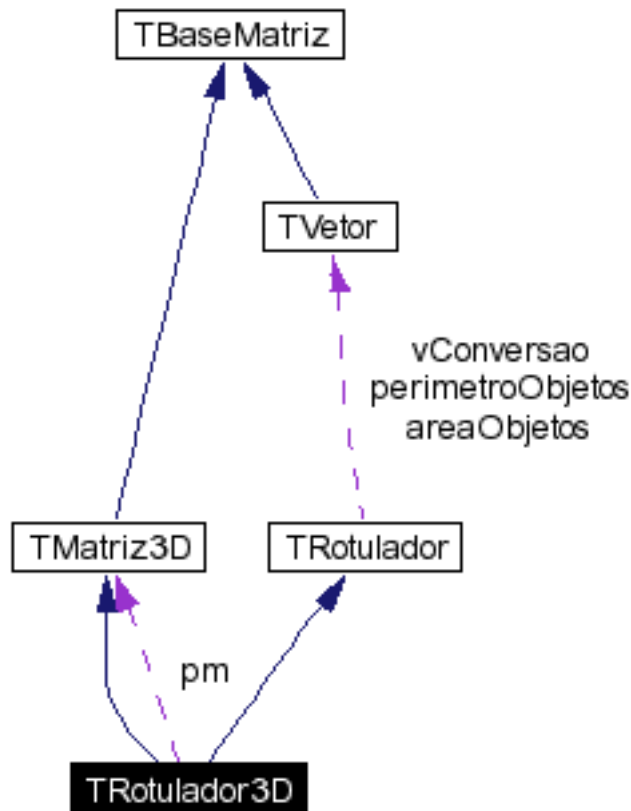
A título ilustrativo, apresenta-se na Figura 51.1 a hierarquia TMatriz da biblioteca LIB_LMPT.

Figura 51.1: Ilustração da hierarquia TMatriz da biblioteca LIB_LMPT.



Apresenta-se na Figura 51.2 a hierarquia da classe TRotulador3D. Observe a relação da classe TRotulador3D com as demais classes.

Figura 51.2: Ilustração da hierarquia da classe TRotulador3D da biblioteca LIB_LMPT.



Apresenta-se na Figura 51.3 as dependências do arquivo TRotulador3D.

51.6 Documentação profissional com sgml/xml (LYX)

Vimos que a documentação de código é um tipo de documentação que é incluída dentro do código, utilizando o formato JAVA_DOC. Que o programa doxygen é utilizado para gerar diversos arquivos html que incluem a documentação de cada arquivo da biblioteca ou programa, e que com o doxygen pode-se gerar saída em outros formatos (como pdf, latex, rtf, xml, manpage).

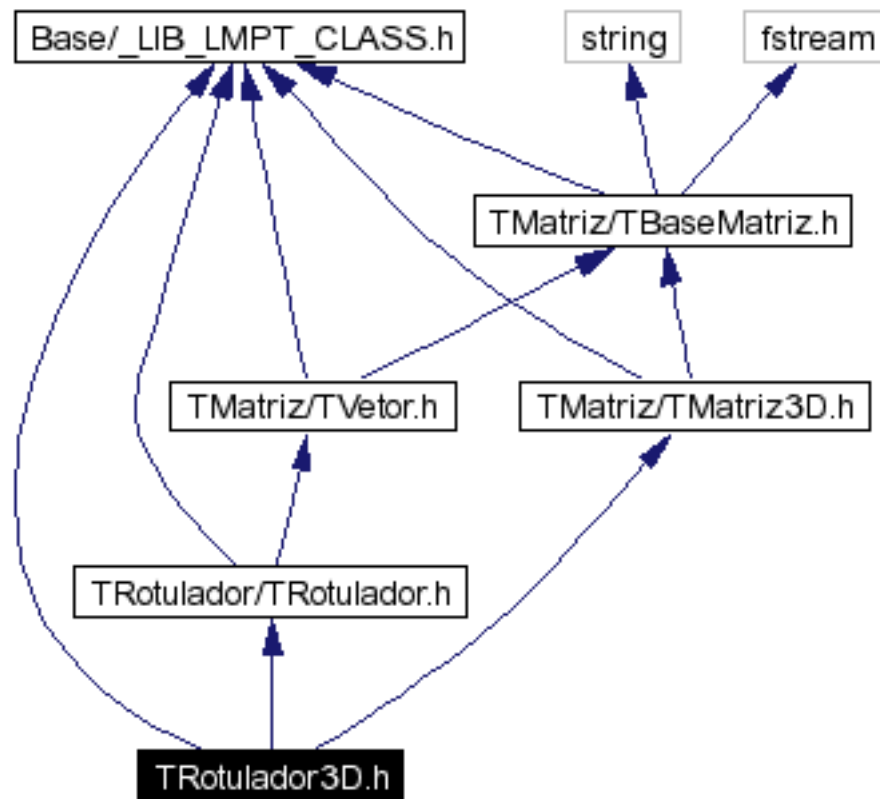
Mas você também vai gerar um manual do usuário e talvez um manual científico. Neste caso como devo proceder para criar estes manuais ?

No mundo Linux, utiliza-se o DOCBOOK.

O docbook é um “document type definition -DTD”, uma especificação de formato de documento. Você pode implementar o docbook usando sgml (standardized general markup language) ou xml (extensible markup language).

É mais ou menos assim: Com docbook você define o formato do manual, com sgml/xml você define a formatação de cada parágrafo.

Figura 51.3: Ilustração das dependências do arquivo TRotulador3D.



Se você instalou todos os pacotes de desenvolvimento de sua distribuição, provavelmente já tem disponibilizados os pacotes necessários. No meu sistema tenho instalados os pacotes:

```

[andre@mercurio Cap4-STL]$
$rpm -qa | egrep docb && rpm -qa | egrep docb
docbook-dtd41-xml-1.0-7
docbook-utils-0.6.9-2
docbook-dtd30-sgml-1.0-10
docbook-dtd41-sgml-1.0-10
docbook-utils-pdf-0.6.9-2
docbook-dtd412-xml-1.0-1
docbook-dtd40-sgml-1.0-11
docbook-dtd31-sgml-1.0-10
docbook-style-dsssl-1.64-3
jadetex-3.11-4
openjade-1.3-17
  
```

Veja detalhes do docbook, sgml e xml no site (<http://www.xml.org>).

Dica preciosa: O programa LyX¹, tem total suporte aos formatos docbook. Desta forma você gera o seu manual do usuário em um programa muito simples de usar (o LyX) e no final pode gerar versões do seu manual nos formatos html, pdf, sgml, txt.

¹Repito, que o LyX foi é utilizado para gerar esta apostila.

Capítulo 52

Sequência de Montagem de Um Programa GNU

Neste capítulo vamos apresentar o desenvolvimento de programas multiplataforma utilizando as ferramentas da GNU.

52.1 Introdução a programação multiplataforma com GNU

Um projeto comum pode ser montado apenas com os arquivos de código (*.h,*.cpp) e o arquivo Makefile. O problema deste formato é a existência de centenas de Unix/Linux, sendo, cada um destes, um pouco diferentes entre sí. Por exemplo, o compilador pode se chamar CC, gcc, c++, g++. Na prática isto implica na necessidade de se corrigir os arquivos Makefile para cada máquina alvo.

Para solucionar este problema, foram desenvolvidos pela GNU, um conjunto de programas que automatizam a geração de código, para as mais variadas plataformas. Dentre estes programas, os mais usuais são o aclocal, o autoheader, o automake, o autoconf e o libtool¹. Estes programas são brevemente descritos neste capítulo.

Observe que ao desenvolver um programa utilizando as ferramentas multiplataforma da GNU, você poderá compilar seu código em qualquer máquina Unix/Linux/Aix/Solaris. Pois um projeto GNU inclui o ./configure, um script de shell, que cria os arquivos Makefile para cada máquina.

Observe na Tabela 52, o diagrama das tarefas executadas pelo programador².

O primeiro programa a ser executado é o aclocal, o mesmo recebe como entrada um conjunto de arquivos de código e um arquivo configure.in, gerando os arquivos aclocal.m4 e acsite.m4.

A seguir, executa-se o ifnames para identificação dos includes e defines em comum. O programador usa o autoscan para gerar um esqueleto inicial do arquivo configure.in, a seguir, o programador usa um editor como o emacs para modificar o arquivo configure.scan, gerando o configure.in. O autoheader é usado para gerar o arquivo config.h.

O programador deve gerar diversos arquivos makefile.am, um para cada diretório e então executar o automake. O automake converte os arquivos makefile.am em makefile, podendo os mesmos ser executados com o programa make.

¹(descrito no Capítulo 47).

²O ambiente de desenvolvimento do kdevelop tem um “wizard” que gera automaticamente todos estes arquivos para você. Veja breve descrição do kdevelop no Capítulo 53.

Finalmente, o programador executa o programa `autoconf`. O `autoconf` gera um script de configuração do programa, usando para isto o arquivo `configure.in`. A saída do `autoconf` é o arquivo `configure`, o mesmo será executado pelo usuário para gerar os `makefiles` específicos para a máquina do usuário. Veja na Tabela 52.2 a sequência executada pelo usuário.

Tabela 52.1: Sequência para montagem de programa GNU.

	Entrada.	Programa executado	Saída.
1	*.h, *.cpp configure.in	aclocal*	aclocal.m4 acsite.m4
2	*.h*, *.cpp	ifnames*	Lista dos defines dos diversos arquivos
3 4	*.h, *.cpp configure.scan	autoscan* usuário	configure.scan configure.in
5	configure.in config.h.top acconfig.h config.h.bot	autoheader*	acsite.m4 config.h.in
6		usuário	Makefile.am
7	Makefile.am	automake*	Makefile.in
8	aclocal.m4 acsite.m4 configure.in	autoconf*	configure

Apresenta-se na Tabela 52.2 a sequência de tarefas executadas pelo usuário. Esta sequência é executada na máquina do usuário para gerar o programa para a sua máquina. Observe que uma das grandes vantagens deste tipo de distribuição, é que se o usuário tem uma máquina Pentium IV, o compilador vai receber instruções para compilar o programa para um Pentium IV, e não para um antigo 386.

Tabela 52.2: Sequência executada pelo usuário.

Entrada	Programa executado	Saída
Makefile.in	./configure	Makefile config.cache config.log
config.h.in Makefile.in	config.status*	config.h Makefile
Makefile	make	Código Compilado
Código compilado	make install	Programa Instalado

52.2 aclocal

O programa aclocal gera o arquivo aclocal.m4 baseado nos dados do arquivo configure.in.

O aclocal procura todos os arquivos *.m4 na path do automake/autoconf, depois procura o arquivo configure.in, copiando todas as macros encontradas para o arquivo aclocal.m4. Ou seja, todas as macros serão copiadas para o arquivo aclocal.m4.

Protótipo e parâmetros do aclocal:

aclocal [opções]

- *-help* *Help do aclocal*
- *-acdir=dir* *Define o diretório*
- *-output=file* *Define nome do arquivo de saída*
- *-verbose* *Modo verbose (detalhado)*

52.3 ifnames

O programa ifnames pesquisa toda a estrutura de diretórios e lista todos os defines dos arquivos *.h e *.cpp. O programa ifnames agrupa todos os defines em um único arquivo o que é útil para gerar o arquivo config.h.in

Protótipo do ifnames:

ifnames [-h][-help][-mdir][-macrodir=dir][-version][file...]

- *-help* *[-h] Mostra help.*
- *-verbose* *[-v] Modo verbose.*
- *-version* *Mostra versão.*

52.4 autoscan

O autoscan pesquisa a estrutura de diretórios e busca arquivos *.h e *.cpp, gerando o arquivo configure.scan. O autoscan extrai informações do código e dos headers, como chamadas de funções.

Protótipo e parâmetros do autoscan:

autoscan [- -macrodir=dir][-help][-verbose]

- *-help*
- *-verbose* *[-v]*
- *-version*
- *-sreaddir* *Diretório a ser escaneado.*

52.4.1 Roteiro do autoscan

1. Execute o autoscan para gerar o configure.scan. O arquivo configure.scan serve de esboço inicial para o arquivo configure.in
2. Corrija o arquivo configure.scan, incluindo ali as macros e definições necessárias.
3. Renomeie o arquivo configure.scan para configure.in. Observe que o arquivo configure.in é um gabarito usado para gerar o arquivo configure final.

- *-version* *Mostra versão do aclocal*

52.5 autoheader

O autoheader pode ser usado para gerar o arquivo config.h.in. O arquivo config.h.in é usado pela macro AC_CONFIG_HEADER(file) para gerar o arquivo config.h. O arquivo config.h contém definições compartilhadas por todos os arquivos do pacote (é um header comum a todo pacote).

Protótipo e parâmetros do autoheader:

```
- -help      [-h]
- -localdir=dir [-l dir]
- -macrodir=dir [-m dir]
- -version
```

52.5.1 Roteiro do autoheader

1. Crie um arquivo config.top com as instruções iniciais do config.h.in.
2. O arquivo acconfig.h contém um conjunto de macros específicas para o sistema, fornecidas pela distribuição do autoconf.
3. Criar o arquivo acconfig.h

```
/* Define if the C++ compiler supports BOOL */
#undef HAVE_BOOL
#undef VERSION
#undef PACKAGE
/* Define if you need the GNU extensions to compile */
#undef _GNU_SOURCE
```

4. Crie um arquivo config.bot com as instruções finais do config.h.in.
5. Execute o autoheader.
6. Edite o arquivo config.h.in gerado pelo autoheader.
7. Inclua no arquivo configure.in a macro AC_CONFIG_HEADER().

52.6 automake

52.6.1 Introdução ao automake

O automake é uma ferramenta para automatizar a criação de makefiles independentes de plataforma. Basicamente o programador escreve um arquivo Makefile.am para cada diretório, o automake lê estes arquivos e o arquivo configure.in e cria os arquivos Makefile.in. O arquivo Makefile.in é um gabarito para geração do Makefile final, isto é, os arquivos Makefile.in serão usados pelo ./configure para geração dos arquivos Makefile finais. A grande vantagem é não precisar reescrever os arquivos Makefile para cada máquina em que você vai compilar o programa.

Vimos no capítulo Make como criar um arquivo makefile simples. O automake cria arquivos makefile complexos.

O automake suporta três tipos de estruturas de diretório

- flat Todos os arquivos estão em um único diretório
- deep Existem vários subdiretórios e arquivos de configuração no diretório base. Dentro do Makefile.am existe a macro SUBDIRS.
- shallow O código primário esta no diretório de mais alto nível e os outros códigos em subdiretórios (usado em bibliotecas).

O automake suporte três tipos de opções de verificação

- 'foreign' Checa somente o necessário (adotado pelo kdevelop).
- 'gnu' É o default, verifica a presença dos arquivos padrões da gnu (INSTALL, NEWS, README, COPYING, AUTHORS, Changelog)
- 'gnits' É um padrão que verifica arquivos adicionais ao padrão gnu. É o mais extenso.

Padronização do formato dos nomes

Os nomes das macros do automake obedecem um padrão uniforme para seus nomes. Se o seu pacote tem um programa chamado meu-programa.1, o mesmo será tratado como meu_programa_1, ou seja substitue - por _ e . por _ .

Protótipo e parâmetros do automake:

automake [opções]

-a [-add-missing]

- -amdir=dir Define diretório.

- -help Ajuda / Help.

- -generate-deps Gera dependências.

- -output-dir=dir Diretório de output/saída.

- -sourcedir=dir Diretório de fonte.

- *-v[-verbose]* Modo *verbose* (detalhado).

- Como é um arquivo makefile, existem instruções como:
PROGRAMS= lista dos programas a serem compilados
EXTRA_PROGRAMS= programas adicionais
bin_PROGRAMS = programas binários
sbin_PROGRAMS = @PROGRAMS@

Exemplo:

```
bin_PROGRAMS = nomeDoPrograma
nomeDoPrograma_SOURCES = nome.h nome.cpp ...
nomeDoPrograma_LDADD = @LIBOBJJS@
```

Dica: A variável @LIBOBJJS@ é copiada do arquivo configure.in para o arquivo makefile (substituição simples). Observe que o valor da variável @LIBOBJJS@ é definida quando o usuário executa o ./configure.

Roteiro do automake

Apresenta-se a seguir o roteiro de execução do automake em um projeto simples.

1. Edite o configure.in, acrescentando as macros:

```
AM_INIT_AUTOMAKE(nome_programa, versao)
AC_REPLACE_FUNCS
LIBOBJJS=listaobjetos da biblioteca
AC_SUBST(LIBOBJJS)
```

2. Cria e edita o arquivo makefile.am

- (a) SUBDIRS = lib_lmpt
EXTRA_DIST = AUTHORS COPYING ChangeLog INSTALL README TODO
ORGANIZATION HOWTO
AUTOMAKE_OPTIONS = foreign
bin_PROGRAMS= nomePrograma
nomePrograma= lista arquivos cpp
nomePrograma_LDADD= @LIBOBJJS@

3. Executa o automake.

Macros do automake

Uma macro realiza um conjunto de operações. Apresenta-se a seguir um conjunto de macros que podem ser incluídas nos arquivos configure.in e makefile.am. Estas macros são usadas para interfacear o automake com o autoconf. As macros do autoconf iniciam com AC e as do automake com AM.

AC_CONFIG_HEADER

O automake requer a macro `AM_CONFIG_HEADER` que é similar a `AC_CONFIG_HEADER`.

AC_PATH_XTRA

Insere as definições do `AC_PATH_XTRA` em cada arquivo `Makefile.am`.

LIBOBJS

Inclue os arquivos `*.o` na lista do `LIBOBJS`.

AC_PROG_RANLIB

Necessário se o pacote compila alguma biblioteca.

AC_PROG_CXX

Necessário se existe código em C++.

AM_PROG_LIBTOOL

Executa o pacote `libtool`.

ALL_LINGUAS

Checa o diretório `po` e os arquivos `.po` para especificações relativas a linguagens.

AUTOMAKE_OPTIONS

Variável especial definida nos arquivos `Makefile.am`. Suporta um subconjunto de variáveis que realizam tarefas específicas.

Exemplos:

```
dist-tarZ Cria uma distribuição do pacote no formato .tar.gz.
```

`AC_CONFIG_AUX_DIR`, `AC_PATH_EXTRA`, `LIBOBJS`, `AC_PROG_RANLIB`,
`AC_PROG_CXX`, `AM_PROG_RANLIB`, `AM_PROG_LIBTOOL`, `ALL_LINGUAS`.

Apresenta-se a seguir um conjunto de macros que podem ser incluídas no arquivo `configure.am` e que são fornecidas pelo automake. Estas macros são usadas para interfacear o automake com o `autoconf`.

AM_INIT_AUTOMAKE(nomePacote,versão)

Inicializa o automake, rodando um conjunto de macros necessárias ao `configure.in`

AM_CONFIG_HEADER

Reconfigurar o arquivo `config.h` levando em conta parâmetros do automake

AM_ENABLE_MULTILIB**AM_FUNC_MKTIME****AM_PROG_CC_STDC**

Se o compilador não está em ANSIC tenta incluir parâmetros para deixá-lo no formato ANSIC

AM_SANITY_CHECK

Verifica se os arquivos `*.o` criados são mais novos que os de código

A macro `AM_INIT_AUTOMAKE`

Existe um conjunto de tarefas que precisam ser realizadas, e que são realizadas automaticamente pela macro `AM_INIT_AUTOMAKE`. As mesmas podem ser executadas manualmente por uma sequência da forma:

Definição das variáveis `PACKAGE` e `VERSION`

Uso da macro `AC_ARG_PROGRAM`

Uso da macro `AM_SANITY_CHECK`

Uso da macro `AC_PROG_INSTALL`

52.6.2 Sentenças para o automake

- Leia o manual do automake, a parte principal, com exemplos não é muito extensa.
- Qualquer variável ou macro definida em um arquivo `makefile.am` sobrescreve variáveis definidas no `configure.in`.
- Você pode incluir variáveis do sistema em seu arquivo `Makefile.am`. Desta forma, pode passar parâmetros para diversos `Makefile.am` de forma simplificada.

– Exemplo:

```
CXXFLAGS = -I${PATH_LIB_LMPT_INCLUDES}
-I${PATH_LIB_IMAGO} -I${PATH_LIB_COILIB}
DEFS = -D__LINUX__ -D__INTEL__ -D__X11__ -D__MESA__
```

52.7 autoconf

52.7.1 Introdução ao autoconf

O autoconf é uma ferramenta que objetiva automatizar a configuração de seu software para a plataforma alvo.

O autoconf inicializa pesquisando as macros instaladas com o pacote autoconf, a seguir verifica a presença do arquivo opcional `acsite.m4` (no diretório de instalação do autoconf) e pelo arquivo `aclocal.m4` (no diretório do programa). O arquivo `aclocal.m4` é criado pelo `aclocal`.

O resultado da execução do autoconf é a criação do arquivo `configure`. O arquivo `configure` será executado pelos usuários para geração dos arquivos `Makefile`, adaptados a plataforma onde o programa vai ser compilado³.

Dica de portabilidade: Se for usar uma biblioteca não portátil, procure criar um módulo separado.

³Lembre-se, quando você faz o download de um programa no formato `.tar.gz`, você primeiro descompacta o arquivo e a seguir executa: `./configure && make && make install`. Observe que quando você executa o `./configure`, uma série de verificações são realizadas em seu sistema.

52.7.2 Protótipo do autoconf

Protótipo e parâmetros do autoconf:

autoconf

-help[-h] *Mostra um help.*

-localdir=dir[-l dir] *Define diretório local.*

-macrodir=dir[-m dir] *Define diretório de macros.*

-version *Mostra versão.*

Vamos apresentar a seguir um exemplo de arquivo `configure.in` e depois descrever rapidamente as macros que você pode incluir no `configure.in`.

52.7.3 Roteiro do autoconf

1. Edite o `configure.in`, acrescentando as macros:

```
AM_INIT_AUTOMAKE(nome_programa,versao)
AC_REPLACE_FUNCS
LIBBOJS=listaobjetos da biblioteca
AC_SUBST(LIBBOJS)
```

52.7.4 Estrutura de um arquivo `configure.in`

O arquivo `configure.in` é usado pelo `autoconf` para montagem do programa `shell configure`. O `configure.in` é composto de um conjunto de macros que serão usadas para testar a configuração-configuração de seu micro (qual compilador esta instalado, sua localização,...).

Basicamente o programador escreve um arquivo `configure.in` contendo:

```
Inclue as macros de inicialização AC_INIT
Inclue as macros de testes
checks for programs
checks for libraries
checks for headers files
checks for typedefs
checks for structures
checks for compiler characteristics
checks for library functions
checks for system services
Inclue as macros de finalização AC_OUTPUT
```

Como visto, você pode utilizar o `autoscan` para gerar um esboço inicial do arquivo `configure.in`.

52.7.5 Exemplo de um arquivo configure.in

Apresenta-se a seguir um exemplo de um arquivo configure.in. Observe que uma linha *dnl* é uma linha de comentário.

O nome do arquivo dentro do AC_INIT(Makefile.am) é usado apenas para verificar o diretório. Os arquivos gerados pelo autoconf estão em AC_OUTPUT.

```
dnl Exemplo de arquivo configure.in
dnl linha de comentário
dnl Process this file with autoconf to produce a configure script.
AC_INIT(Makefile.am)
AM_CONFIG_HEADER(config.h)
AM_INIT_AUTOMAKE(lib_lmpt,0.1)
dnl Checks for programs.
AC_PROG_CC
AC_PROG_CXX
AC_PROG_RANLIB
dnl Checks for libraries.
dnl Checks for header files.
dnl Checks for typedefs, structures, and compiler characteristics.
dnl Checks for library functions.
AC_OUTPUT(Makefile lib_lmpt/Makefile lib_lmpt/include/Base/Makefile
lib_lmpt/include/Makefile lib_lmpt/source/Base/Makefile lib_lmpt/source/Makefile)
```

Dica: Não deixar espaços entre o nome da macro e os paranteses.

52.7.6 Macros do autoconf

Lista-se a seguir um conjunto de macros que podem ser utilizadas para testar a presença de algo. Estas macros fazem parte do autoconf e devem ser incluídas no arquivo configure.in. Observe que iniciam com AC se forem macros do autoconf e AM se forem macros do automake.

As principais macros a serem incluídas no arquivo configure.in são a AC_INIT e AC_OUTPUT.

AC_INIT(arquivo)

- Processa os parâmetros da linha de comando, e
- Pesquisa a estrutura de diretórios pelos arquivos *.h e *.cpp.
- O nome do arquivo dentro do AC_INIT é usado apenas para verificar se o diretório esta correto.

AC_OUTPUT([arquivo],[comandos extras],[comandos inicialização]))

- Macro que gera os arquivos de saída do comando autoconf. Gera os arquivos Makefile.in e configure. Observe que você pode passar comandos extras e comandos de inicialização. Ou seja, no AC_OUTPUT serão colocados os nomes dos arquivos que serão gerados.

Mas existem muitas outras macros que estão disponíveis e que você pode usar, macros para testes em geral, para pesquisar se determinado programa/biblioteca esta instalado(a), se determinadas funções estão disponíveis, entre outros. Lista-se a seguir as macros mais utilizadas.

Testes de uso geral

AC_CONFIG_AUX_DIR

Configurações auxiliares.

AC_OUTPUT_COMMANDS

Execução de comandos de shell adicionais.

AC_PROG_MAKE_SET

Usado para definir a variável MAKE=make.

AC_CONFIG_SUBDIRS

Rodar o configure em cada um dos subdiretórios incluídos nesta listagem.

AC_PREFIX_DEFAULT(prefix)

Seta o prefixo default para instalação (o padrão é /usr/local).

AC_PREFIX_PROGRAM(program)

Se o usuário não entrar com -prefix, procura por um prefixo na PATH

AC_PREREQ(version)

Informa a versão do autoconf que deve ser utilizada, se a versão instalada for anterior emite mensagem de erro.

AC_REVISION(revision-info)

Inclue mensagem no início dos arquivos informando o número da revisão.

Pesquisando programas

AC_PROG_CPP

Seta a variável CPP com o nome do pré-processador C existente.

AC_PROG_CXX

Verifica se já foi definida a variável CXX ou CCC (nesta ordem). Se não definida procura o compilador C++ e seta a variável CXX. Se o compilador for da GNU, seta a variável GXX=yes. Se CXXFLAGS não foi definido seta como -g -o2.

AC_PROG_CC

Identifica o compilador C, e define a variável CC com o nome do compilador encontrado. Adicionalmente se encontrou o GNU-GCC define a variável GCC=yes caso contrário GCC=.

AC_PROG_CXXCPP

Seta a variável CXXCPP com o nome do pre-processador C++.

AC_PROG_INSTALL

Seta a variável INSTALL com a path compatível com o programa de instalação BSD.

AC_PROG_LN_S

Verifica se o sistema aceita links simbólicos e seta a variável LNS como ln -s.

AC_PROG_RANLIB

Seta a variável RANLIB se o ranlib esta presente.

AC_CHECK_PROG(variável,programa,açãoTrue,açãoFalse)

Checa a existência da variável e do programa, se ok realiza ação true se false realiza ação false.

AC_CHECK_PROGS(variável,programas,açãoTrue,açãoFalse)

Checa a existência da variável e dos programas, se ok realiza ação true se false realiza ação false.

Pesquisando bibliotecas**AC_CHECK_LIB**(biblioteca,função,açãoTrue,açãoFalse)

Verifica se a função pertence a biblioteca.

AC_HAVE_LIBRARY(biblioteca,açãoTrue,açãoFalse)

Verifica se a biblioteca existe.

AC_SEARCH_LIB(função,listaDeBibliotecas,açãoTrue,açãoFalse)

Pesquisa a função no conjunto de bibliotecas listadas.

AC_TRY_LINK**AC_TRY_LINK_FUNC****AC_COMPILE_CHECK****Pesquisando funções****AC_CHECK_FUNC**(função,açãoTrue,açãoFalse)

Verifica se a função existe, e executa o comando de shell.

AC_CHECK_FUNCS(função...,açãoTrue,açãoFalse)

Verifica se a função existe, e executa o comando de shell.

AC_REPLACE_FUNCS(função...)

Adiciona a função com o nome função.o a variável LIBOBJS.

Pesquisando arquivos *.h e *.cpp**AC_CHECK_HEADER**(header,açãoTrue,açãoFalse)

Verifica a existencia do header, se existe executa açãoTrue.

AC_CONFIG_HEADER(header_a_ser_criado)

Arquivo a ser criado com os #defines. Substitue @DEFS@ por -DHAVE_CONFIG_H, o nome padrão para o arquivo é config.h. Usado para criar o arquivo config.h com os header comuns ao pacote.

AC_CHECK_FILE(arquivo,açãoTrue,açãoFalse)

Checa se o arquivo existe.

AC_CHECK_FILES(arquivos,açãoTrue,açãoFalse)

Checa um conjunto de arquivos

AC_TRY_CPP(includes[,açõesTrue,açõesFalse])

Procura pelos arquivos include, se localizou realiza a ação true, caso contrário a ação false.

AC_EGREP_HEADER(padrãoPesquisa,headerFile,ação)

Se a pesquisa do padrão no arquivo header foi ok, realiza a ação.

AC_EGREP_CPP(padrãoPesquisa,cppFile,ação)

Se a pesquisa foi ok, realiza a ação.

AC_TRY_COMPILE(includes,corpoDaFunção,açãoTrue,açãoFalse)

Cria um programa de teste, com a função especificada para verificar se a função existe.

PS: Arquivos de headers especificados pelas macros HEADERS..., geralmente não são instalados, e os headers listados em ..._SOURCES não podem ser incluídos nos ..._HEADERS.

Rodando programas de teste

AC_TRY_RUN(programa,açãoTrue,açãoFalse)

Tenta rodar o programa, se ok realiza açãoTrue.

Pesquisando estruturas: Veja manual do autoconf.

Pesquisando typedefs: Veja manual do autoconf.

Pesquisando características do compilador C

AC_C_CONST

Verifica se o compilador suporta variáveis const.

AC_C_INLINE

Verifica se o compilador suporta funções inline.

AC_CHECK_SIZEOF(tipo[,tamanho])

Ex: AC_CHECK_SIZEOF(int *).

52.7.7 Como aproveitar os resultados das pesquisas realizadas pelo autoconf

Você coloca um conjunto de macros no arquivo configure.in para testar o seu programa. Os resultados das pesquisas realizadas pelo autoconf podem ser salvas. Como exemplos, definições de diretrizes de pré-processador, definição de variáveis de shell. Você pode salvar os resultados em caches, ou imprimir os resultados das pesquisas na tela. As macros abaixo mostram como fazer:

AC_DEFINE(variável,valor,descrição)

Cria variável define. Ex: AC_DEFINE(EQUATION, \$a > \$b)

AC_SUBST(variável)

Cria uma variável de saída a partir de variável de shell.

Define o nome de uma variável que deverá ser substituída nos arquivos Makefile.

AC_SUBST_FILE(variável)

O mesmo que acima.

AC_CACHE_VAL(cache-id,comando_do_id)

Veja manual do autoconf.

AC_CACHE_CHECK

Verifica o cache.

AC_CACHE_LOAD

Carrega algo do cache.

AC_CACHE_SAVE

Salva algo no cache.

AC_MSG_CHECKING(descrição)

Informa o usuário que executou o ./configure o que esta fazendo.

AC_MSG_RESULT(descrição_do_resultado)

Normalmente uma mensagem com o resultado do que foi checado.

AC_MSG_ERROR(descrição_da_mensagem_de_erro)

Emite uma mensagem de erro.

AC_MSG_WARN(descrição_da_mensagem_de_warning)

Emite uma mensagem de warning.

Como definir variáveis no configure.in e usar no makefile.am

Apresenta-se a seguir um exemplo de definição de variável no arquivo configure.in que vai ser usada no arquivo makefile.am.

No arquivo configure.in

```
SUBDIRS= "source doc"  
AC_SUBST(SUBDIRS)
```

No arquivo makefile.am

```
##linha de comentário  
VARIABLE=valor  
PROG_SOURCES= prog.cpp $(VARIABLE)  
SUBDIRS= @SUBDIRS@
```

52.7.8 Variáveis definidas no arquivo `configure.in` e que serão substituídas no arquivo `Makefile`

Além de testar o sistema como um todo, o `autoconf` permite a inclusão de definições no arquivo `configure.in` que serão substituídas nos arquivos `Makefile`. Desta forma, você não precisa ficar conferindo se uma variável definida em cada arquivo `Makefile.am` esta coerente. Basta definir a variável no `configure.in`, que a mesma será copiada para cada arquivo `Makefile`. Apresenta-se a seguir uma lista resumida das variáveis que são definidas no `configure.in` para uso do `makefile.am`.

bindir Diretório de instalação do executável.

configure_input Para incluir um comentário informando que o arquivo foi gerado pelo `autoconf`.

datadir Diretório para instalação dos arquivos `ready-only`.

exec_prefix Prefixo dos arquivos executáveis que dependem da arquitetura.

includedir Diretório para instalação dos arquivos de headers.

infodir Diretório para instalação dos arquivos de documentação.

libdir Diretório para instalação dos objetos das bibliotecas.

libexecdir Diretório para instalação de executáveis que outros programas executam.

prefix Prefixo para arquivos dependentes de plataforma.

CXXCOMPILE

Comando para compilação de códigos C++, normalmente setado como:
`CXXCOMPILE= $(CXX) $(DEFS) $(INCLUDES)
$(AM_CPPFLAGS) $(CPPFLAGS) $(AM_CXXFLAGS) $(CXXFLAGS)`

CXX

Nome do compilador C++.

DEFS

Opção `-D` para compiladores C.

CFLAGS

Opções de debugagem e otimização para compiladores C.

CPPFLAGS

Diretório com arquivos headers, e outros parâmetros para o preprocessor e compilador C.

CXXLINK

Comando para linkagem de programas C++, normalmente setado como:
`CXXLINK=$(CXXLD) $(AM_CXXFLAGS) $(CXXFLAGS) $(LDFLAGS)`

CXXFLAGS

Opções de debugagem e otimização para compiladores C++.

LDFLAGS

Opções para o linker.

LIBS

Opções -l e -L para o linker.

Você pode incluir em seus arquivos makefile.am variáveis definidas no shell. Ou seja, antes de executar o `./configure && make && make install`, o usuário define algumas variáveis em seu shell (ex: `PATH_INCLUDE_LIB_LMPT=/usr/include/lib_lmpt`).

Escrevendo suas macros para utilização com o autoconf

Apresentou-se acima um conjunto de macros disponíveis no pacote autoconf para uso no arquivo autoconf.in. Adicionalmente, você pode construir suas próprias macros e incluir no arquivo configure.in. Veja a seção manual do autoconf.

Variáveis relacionadas ao autoconf

Veja a seção manual do autoconf.

Variáveis de ambiente setadas pelo autoconf Veja a seção manual do autoconf.

Variáveis geradas pelo autoconf Veja a seção manual do autoconf.

Defines usados pelo autoconf (e que você não deve usar nos seus programas)

Lista das macros do autoconf (macros que podem ser incluídas no configure.in)

PS: Observe que no arquivo configure.in existem macros do autoconf (iniciadas com `AC_`) e do automake (iniciadas com `AM_`), mostrando uma interdependência do automake e do autoconf .

52.8 autoreconf

Pacote utilizado para reconstruir aquilo que o autoconf construiu, no caso de alterações na instalação.

O autoreconf atualiza os scripts configure.

Protótipo e parâmetros do autoreconf:

autoreconf

-help [*-h*]

-force

-localdir=dir [*-l dir*]

-macrodir=dir [*-m dir*]

-verbose [*-v*]

-version

52.9 ./configure

A execução do ./configure gera:

- Um conjunto de arquivos Makefile.
- Um arquivo de headers com defines.
- Um arquivo config.status.
- Um arquivo de shell que salva os dados de configuração em um cache.
- Um arquivo config.log com as informações da execução do ./configure.

52.10 Como incluir instruções do libtool em seu pacote gnu

Você precisa incluir os arquivos:

- 'config.guess' Nomes canônicos.
- 'config.sub' Validação de nomes.
- 'ltmain.sh' Implementa funções básicas do libtool.

Você pode usar o programa libtoolize. O programa libtoolize adiciona aos seus arquivos as instruções para dar suporte ao libtool, adicionando os arquivos 'config.guess' 'config.sub' 'ltmain.sh'.

Protótipo e parâmetros do libtoolize:

```
libtoolize [opções]
```

- automake
- copy Cópia os arquivos e não os links para o diretório.
- n Não modifica os arquivos, apenas mostra as modificações.
- force Força a substituição dos arquivos existentes do libtool.
- help Ajuda.

52.10.1 Exemplo de arquivo makefile.am usando o libtool

```
Exemplo:
bin_PROGRAMS = prog prog.debug
#Gera o programa
prog_SOURCES = *.cpp
prog_LDADD= libNome.a      ##'-dlopen''
#Gera o programa com debug
prog_debug_SOURCES = *.cpp
prog_debug_LDADD= libNome.a      ##'-dlopen''
prog_debug_LDFLAGS= -static
```

52.10.2 Exemplo de arquivo configure.in usando o libtool

Exemplo, acrescentar as macros:

```
##suporte do autoconf ao libtool
AC_PROG_LIBTOOL
##suporte do automake ao libtool
AM_PROG_LIBTOOL
##suporte a bibliotecas dinâmicas (?)
AC_LIBTOOL_DLOPEN
```

Sentenças:

- Leia os capítulos “Using Libttol” e “integrating libttol” do manual do libtool.
- Basicamente o libtool é suportado pela variável LTLIBRARIES.

52.11 Exemplo Completo

Apresenta-se a seguir os arquivos do programa LIB_LMPT. A estrutura de diretórios é da forma

```
LIB_LMPT
LIB_LMPT/lib_lmpt (Arquivos main.cpp, teste.cpp, teste.h)
LIB_LMPT/lib_lmpt/source/base (Arquivos TOperacao.cpp, TMath.cpp)
LIB_LMPT/lib_lmpt/include/base (Arquivos TOperacao.h, TMath.h)
LIB_LMPT/lib_lmpt/docs
```

LIB_LMPT/Makefile.am

```
SUBDIRS = lib_lmpt
EXTRA_DIST = AUTHORS COPYING ChangeLog INSTALL README TODO
ORGANIZATION HOWTO
AUTOMAKE_OPTIONS = foreign
```

LIB_LMPT/Madefile.dist

```
default: all
dist: @echo "This file is to make it easier for you to create all you need"
aclocal
autoheader
# use --include-deps, if you want to release the stuff. Don't use it for yourself
automake --include-deps
autoconf
touch stamp-h.in
LIST='find ./po -name "*.po"'; \
for i in $$LIST; do \
file2='echo $$i | sed -e "s#\.\po#\.\gmo#"'; \
```

```
msgfmt -o $$file2 $$i; \
done
rm -f Makefile.dist
all: aclocal autoheader automake autoconf
```

LIB_LMPT/acconfig.h

```
/* Define if the C++ compiler supports BOOL */
#undef HAVE_BOOL
#undef VERSION
#undef PACKAGE
/* Define if you need the GNU extensions to compile */
#undef _GNU_SOURCE
```

LIB_LMPT/aclocal.m4

Arquivo grande contendo um conjunto de macros.

LIB_LMPT/config.cache

```
# This file is a shell script that caches the results of configure
# tests run on this system so they can be shared between configure
# scripts and configure runs. It is not useful on other systems.
# If it contains results you don't want to keep, you may remove or edit it.
#
# By default, configure uses ./config.cache as the cache file,
# creating it if it does not exist already. You can give configure
# the --cache-file=FILE option to use a different cache file; that is
# what configure does when it calls configure scripts in
# subdirectories, so they share the cache.
# Giving --cache-file=/dev/null disables caching, for debugging configure.
# config.status only pays attention to the cache file if you give it the
# --recheck option to rerun configure.
#
ac_cv_path_install=${ac_cv_path_install='/usr/bin/install -c'}
ac_cv_prog_CC=${ac_cv_prog_CC='gcc'} ac_cv_prog_CXX=${ac_cv_prog_CXX='c++'}
ac_cv_prog_RANLIB=${ac_cv_prog_RANLIB='ranlib'}
ac_cv_prog_cc_cross=${ac_cv_prog_cc_cross='no'}
ac_cv_prog_cc_g=${ac_cv_prog_cc_g='yes'}
ac_cv_prog_cc_works=${ac_cv_prog_cc_works='yes'}
ac_cv_prog_cxx_cross=${ac_cv_prog_cxx_cross='no'}
ac_cv_prog_cxx_g=${ac_cv_prog_cxx_g='yes'}
ac_cv_prog_cxx_works=${ac_cv_prog_cxx_works='yes'}
ac_cv_prog_gcc=${ac_cv_prog_gcc='yes'}
ac_cv_prog_gxx=${ac_cv_prog_gxx='yes'}
ac_cv_prog_make_make_set=${ac_cv_prog_make_make_set='yes'}
```

LIB_LMPT/config.h.in

```

/* config.h.in. Generated automatically from configure.in by autoheader. */
/* Name of package */
#undef PACKAGE
/* Version number of package */
#undef VERSION
LIB_LMPT/config.h
/* config.h. Generated automatically by configure. */
/* config.h.in. Generated automatically from configure.in by autoheader. */
/* Name of package */
#define PACKAGE "lib_lmpt"
/* Version number of package */
#define VERSION "0.1"

```

LIB_LMPT/config.log

This file contains any messages produced by compilers while running configure, to aid debugging if configure makes a mistake.

```

configure:561:
checking for a BSD compatible install
configure:614: checking whether build environment is sane
configure:671: checking whether make sets ${MAKE}
configure:717: checking for working aclocal
configure:730: checking for working autoconf
configure:743: checking for working automake
configure:756: checking for working autoheader
configure:769: checking for working makeinfo
configure:786: checking for gcc
configure:899: checking whether the C compiler (gcc ) works
configure:915: gcc -o conftest conftest.c 1>&5
configure:941: checking whether the C compiler (gcc )
is a cross-compiler
configure:946: checking whether we are using GNU C
configure:974: checking whether gcc accepts -g
configure:1010: checking for c++
configure:1042: checking whether the C++ compiler (c++ ) works
configure:1058: c++ -o conftest conftest.C 1>&5
configure:1084: checking whether the C++ compiler (c++ ) is a cross-compiler
configure:1089: checking whether we are using GNU C++
configure:1117: checking whether c++ accepts -g
configure:1151: checking for ranlib

```

LIB_LMPT/config.status

```

#! /bin/sh # Generated automatically by configure.

```



```
# Run this file to recreate the current configuration.
```

LIB_LMPT/configure

```
#O arquivo configure é um arquivo de shell grande ( 50k)
#criado pelo autoconf,e que quando executado
#faz centenas de testes no sistema do usuário e cria os arquivos Makefile.
# Guess values for system-dependent variables and create Makefiles.

# Generated automatically using autoconf version 2.13
```

LIB_LMPT/lib_lmpt/Makefile.am

```
bin_PROGRAMS = lib_lmpt
lib_lmpt_SOURCES = TTeste.cpp main.cpp
lib_lmpt_LDADD = ./source/Base/libBase.a
SUBDIRS = include source
EXTRA_DIST = main.cpp TTeste.cpp TTeste.h
```

LIB_LMPT/lib_lmpt/include/Makefile.am

```
SUBDIRS = Base
```

LIB_LMPT/lib_lmpt/include/Base/Makefile.am

```
EXTRA_DIST = TMath.h TOperacao.h
```

LIB_LMPT/lib_lmpt/source/Makefile.am

```
SUBDIRS = Base
```

LIB_LMPT/lib_lmpt/source/Base/Makefile.am

```
noinst_LIBRARIES = libBase.a
libBase_a_SOURCES = TOperacao.cpp TMath.cpp
EXTRA_DIST = TMath.cpp TOperacao.cpp
```

LIB_LMPT/lib_lmpt/docs/en/Makefile.am

```
EXTRA_DIST = index.html index-1.html index-2.html index-3.html
index-4.html index-5.html index-6.html
```


Capítulo 53

Ambientes de Desenvolvimento no Linux

53.1 Kdevelop

53.1.1 O que é o kdevelop ?

Um ambiente de desenvolvimento completo para o Linux.

É um ambiente de desenvolvimento moderno. Permite visualizar os arquivos e as classes de diferentes formas. Tem syntax-highlight, documentação em html, e muito mais.

53.1.2 Onde encontrar ?

No site : <http://www.kdevelop.org>.

53.1.3 Como instalar ?

Você pode baixar o pacote rpm (ou compatível com sua versão do GNU/Linux) e instalar em sua máquina:

```
Exemplo:  
rpm -Uvh kdevelop-versao.rpm.
```

Veja na Figura 53.1 a tela do kdevelop.

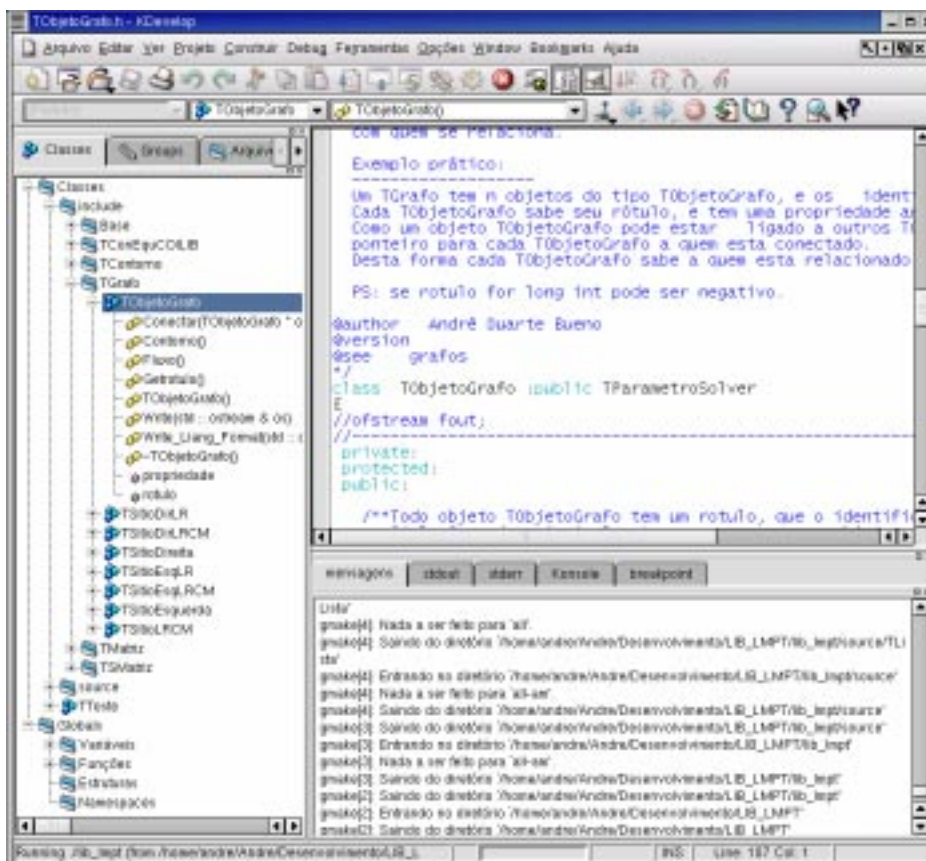


Figura 53.1: A tela do kdevelop (<http://www.kdevelop.org>).

Capítulo 54

Introdução ao Controle de Versões Com o CVS

Neste capítulo vamos apresentar o CVS, um sistema para controle das versões de seu programa ou projeto. Vamos ver o que é o cvs, os comandos e a sequência de trabalho.

Este capítulo foi escrito usando as referencias ([Cederqvist, 1993, Nolden and Kdevelop-Team, 1998, Hughs and Hughes, 1997, Kurt Wall, 2001]).

54.1 O que é o CVS¹?

CVS é um sistema de controle de versões (Concurrent Versions System).

- Com CVS você pode gerenciar diferentes versões de um programa (ou projeto).
- Pode atualizar, adicionar e eliminar arquivos e diretórios ao programa.
- Pode criar ramificações de um projeto.
- Múltiplos programadores podem trabalhar ao mesmo tempo no mesmo projeto.
- Informações recentes sobre o CVS você encontra no site (<http://www.cvshome.org/>).

O que é o repositório?

É um diretório com todos os arquivos e subdiretórios do projeto. Adicionalmente, contém arquivos criados pelo programa cvs para o gerenciamento das versões.

O que é uma versão, um tag, um release ?

Todo arquivo tem uma **versão** que é automaticamente definida pelo cvs. Um **tag** é um nome simbólico dado a uma determinada versão do projeto, pode ser usado para delimitar etapas do desenvolvimento de um projeto. Um **release** é uma versão definitiva de todos os arquivos do projeto.

¹O que o CVS não é ? CVS não é um sistema para construção do soft. Não substitue o gerenciamento do soft. Não substitue a necessidade de comunicação entre o grupo de desenvolvimento. Não serve para testar o soft.

O que é um branch (ramo)?

Um branch (ramo) é usado para dividir um projeto. Normalmente existe o ramo mestre e os ramos secundários.

54.2 Comandos do cvs

Veja a seguir o protótipo do programa cvs. Observe que você passa um conjunto de opções para o cvs; depois, o nome do comando a ser executado e um conjunto de argumentos relativos ao comando.

Protocolo:

```
cvs [cvs-options] command [command-options-and-arguments]
```

Os principais comandos do cvs são o **cvs checkout** que baixa os arquivos do repositório para seu local de trabalho, o **cvs update** que atualiza os arquivos do local de trabalho, e o **cvs commit**, que devolve ao repositório os arquivos que você modificou.

Lista-se a seguir a saída do comando **cvs - -help-options** que mostra a lista de opções do programa cvs.

Listing 54.1: Saída do comando: cvs -help-options

```
[andre@mercurio cvs]$ cvs --help-options
CVS global options (specified before the command name) are:
-H          Displays usage information for command.
-Q          Cause CVS to be really quiet.
-q          Cause CVS to be somewhat quiet.
-r          Make checked-out files read-only.
-w          Make checked-out files read-write (default).
-l          Turn history logging off.
-n          Do not execute anything that will change the disk.
-t          Show trace of program execution -- try with -n.
-v          CVS version and copyright.
-T tmpdir  Use 'tmpdir' for temporary files.
-e editor   Use 'editor' for editing log information.
-d CVS_root Overrides $CVSRROOT as the root of the CVS tree.
-f          Do not use the ~/.cvsrc file.
-z #        Use compression level '#' for net traffic.
-x          Encrypt all net traffic.
-a          Authenticate all net traffic.
-s VAR=VAL  Set CVS user variable.
```

Lista-se a seguir a saída do comando **cvs - -help-commands** o mesmo apresenta a lista de comandos do cvs.

Listing 54.2: Saída do comando: cvs -help-commands

```
CVS commands are:
  add          Add a new file/directory to the repository
  admin        Administration front end for rcs
  annotate      Show last revision where each line was modified
  checkout     Checkout sources for editing
  commit      Check files into the repository
  diff        Show differences between revisions
```

edit	Get ready to edit a watched file
editors	See who is editing a watched file
export	Export sources from CVS, similar to checkout
history	Show repository access history
import	Import sources into CVS, using vendor branches
init	Create a CVS repository if it doesn't exist
kserver	Kerberos server mode
log	Print out history information for files
login	Prompt for password for authenticating server
logout	Removes entry in .cvspass for remote repository
pserver	Password server mode
rannotate	Show last revision where each line of module was modified
rdiff	Create 'patch' format diffs between releases
release	Indicate that a Module is no longer in use
remove	Remove an entry from the repository
rlog	Print out history information for a module
rtag	Add a symbolic tag to a module
server	Server mode
status	Display status information on checked out files
tag	Add a symbolic tag to checked out version of files
unedit	Undo an edit command
update	Bring work tree in sync with repository
version	Show current CVS version(s)
watch	Set watches
watchers	See who is watching a file

Como alguns comandos podem ser repetidos com frequência, os mesmos possuem sinônimos. A listagem a seguir apresenta estes sinônimos.

Listing 54.3: Saída do comando: cvs-help-synonyms

```
[andre@mercurio cvs]$ cvs --help-synonyms
CVS command synonyms are:
  add          ad new
  admin        adm rcs
  annotate      ann
  checkout     co get
  commit       ci com
  diff         di dif
  export       exp ex
  history      hi his
  import       im imp
  log          lo
  login        logon lgn
  rannotate    rann ra
  rdiff        patch pa
  release      re rel
  remove       rm delete
  rlog         rl
  rtag         rt rfreeze
  status       st stat
  tag          ta freeze
  update       up upd
  version      ve ver
```

Para obter um help específico sobre um determinado comando use o comando: **cvs -H comando**.

54.3 Sequência de trabalho

Apresenta-se nas seções que seguem os comandos e exemplos de uso do cvs.

Primeiro vamos criar o repositório, a seguir vamos importar um projeto antigo (que já existia) para dentro do repositório. Definido o repositório e importado o projeto, podemos iniciar o uso efetivo do cvs. Vamos criar um diretório de trabalho e com o comando checkout copiar os arquivos do repositório para dentro de nosso diretório de trabalho. Vamos aprender a adicionar novos arquivos e diretórios ao projeto. Finalmente, vamos devolver para o repositório os arquivos modificados com o comando commit.

54.3.1 Roteiro para criar um repositório

1. Setar a variável CVSROOT no arquivo profile (ou no arquivo `~/bash_profile`):

```
CVSROOT=/home/REPOSITORY
export CVSROOT
```

Se estiver usando o cshel

```
setenv CVSROOT = /home/REPOSITORY
```

2. A seguir, você deve criar o diretório onde o repositório vai ser armazenado (se necessário, como root):

```
mkdir /home/REPOSITORY
```

- Todos os usuários que vão usar o cvs devem ter acesso a este diretório. A dica é criar um grupo de trabalho com permissão de leitura e escrita ao diretório do repositório.
3. Você pode criar um grupo cvs, adicionar ao grupo cvs os usuários que terão acesso ao repositório e mudar as permissões de acesso ao repositório.

```
chown -R cvs /home/REPOSITORY
chmod g+rwX /home/REPOSITORY
```

4. O comando **init** inicializa o uso do cvs, adicionando ao diretório do repositório (`/home/REPOSITORY`) alguns arquivos de controle do programa cvs.

```
cvs init
```

Dê uma olhada no diretório `/home/REPOSITORY`, observe que foi criado o subdiretório `/home/REPOSITORY/REPOSITORY`. Este subdiretório contém os arquivos de administração do cvs.

- Os arquivos com `*,v` são read-only.
- ²A variável CVSUMASK é usada para controlar a forma como os arquivos e diretórios são criados. Consulte um manual de Linux/Unix/Mac Os X para maiores detalhes.

54.3.2 Para importar os arquivos de seu projeto antigo para dentro do repositório

Você provavelmente já tem um diretório com projetos antigos e com arquivos de programação (*.h, *.cpp). O comando **import** copia o seu diretório para dentro do repositório.

Protótipo:

```
cd path_completa_projeto_antigo
cvs import -m "mensagem" path_proj_no_repositorio nome_release nome_tag
```

-m "msg" É uma mensagem curta contendo informação sobre o projeto.

path_proj_no_repositorio É a path para o diretório do projeto no repositório.

nome_release É o nome do release inicial.

nome_tag Informa o tag inicial do projeto (normalmente = start).

Vamos adicionar ao repositório o projeto exemplo-biblioteca-gnu localizado, em minha máquina, no diretório:

```
~/ApostilaProgramacao/Exemplos/Cap-GNU/biblioteca.
```

```
cd ~/ApostilaProgramacao/Exemplos/Cap-GNU/biblioteca
cvs import -m "Exemplo de biblioteca usando ferramentas gnu"
exemplo-biblioteca-gnu R1 start
```

A saída gerada pelo comando import é apresentada na listagem a seguir. Observe que a letra N indica um arquivo novo, a letra I um arquivo ignorado (arquivos *.bak *.~ são ignorados pelo cvs). A biblioteca recebe um L de library.

Listing 54.4: Saída do comando: cvs -import

```
[andre@mercurio biblioteca]$ cvs import -m "Exemplo de biblioteca usando
ferramentas gnu" exemplo-biblioteca-gnu R1 start
N exemplo-biblioteca-gnu/e87-Polimorfismo.cpp
N exemplo-biblioteca-gnu/e87-Programa.cpp
N exemplo-biblioteca-gnu/e87-TCirculo.cpp
I exemplo-biblioteca-gnu/doxygen.config.bak
N exemplo-biblioteca-gnu/makefile
N exemplo-biblioteca-gnu/e87-TCirculo.h
N exemplo-biblioteca-gnu/doxygen.config
N exemplo-biblioteca-gnu/uso-makefile
N exemplo-biblioteca-gnu/e87-PolimorfismoStatic.cpp
N exemplo-biblioteca-gnu/e87-TElipse.cpp
N exemplo-biblioteca-gnu/e87-TElipse.h
N exemplo-biblioteca-gnu/e87-PolimorfismoDinamic.cpp
N exemplo-biblioteca-gnu/Makefile
N exemplo-biblioteca-gnu/e87-TPonto.cpp
N exemplo-biblioteca-gnu/e87-TPonto.h
N exemplo-biblioteca-gnu/e87-Polimorfismo
I exemplo-biblioteca-gnu/e87-Polimorfismo.cpp~
N exemplo-biblioteca-gnu/makefile-libtool
cvs import: Importing /home/REPOSITORY/exemplo-biblioteca-gnu/.libs
N exemplo-biblioteca-gnu/.libs/libTPonto.al
```

```
L exemplo-biblioteca-gnu/.libs/libTPonto.la
```

```
No conflicts created by this import
```

Você pode executar o comando `ls /home/REPOSITORY` ou `tree /home/REPOSITORY` para ver como os arquivos foram importados para dentro do repositório.

Listing 54.5: Como fica o repositório após a importação

```
/home/REPOSITORY/
|-- CVSRROOT
|   |-- modules
|   |-- notify
|   |-- .....
|   '-- verifysmg,v
'-- exemplo-biblioteca-gnu
    |-- Makefile,v
    |-- doxygem.config,v
    |-- doxygem.configold,v
    |-- e87-Polimorfismo,v
    |-- e87-Polimorfismo.cpp,v
    |-- e87-PolimorfismoDinamic.cpp,v
    |-- e87-PolimorfismoStatic.cpp,v
    |-- e87-Programa.cpp,v
    |-- e87-TCirculo.cpp,v
    |-- e87-TCirculo.h,v
    |-- e87-TElipse.cpp,v
    |-- e87-TElipse.h,v
    |-- e87-TPonto.cpp,v
    |-- e87-TPonto.h,v
    |-- makefile,v
    |-- makefile-funciona,v
    |-- makefile-libtool,v
    |-- makefile-ok,v
    '-- uso-makefile,v
```

Dica: Depois de importar seus projetos para dentro do repositório, faça um backup dos projetos (`tar -cvzf NomeProjeto.tar.gz NomeProjeto`) e remova os arquivos do projeto (`rm -fr NomeProjeto`). Desta forma você elimina a possibilidade de trabalhar acidentalmente nos arquivos de seu projeto em vez de trabalhar com os arquivos do repositório.

54.3.3 Para baixar o projeto

O nosso repositório já foi criado, já definimos um grupo de trabalho e já copiamos para dentro do repositório um projeto. Agora vamos iniciar o uso efetivo do cvs.

Para copiar os arquivos de dentro do repositório para o diretório onde você deseja trabalhar, usa-se o comando **checkout**. Veja na listagem a seguir o protótipo e os parâmetros do comando **checkout**.

Listing 54.6: Saída do comando: `cvs -H checkout`

```
[andre@mercurio cvs]$ cvs -H checkout
Usage:
  cvs checkout [-ANPRcflnps] [-r rev] [-D date] [-d dir]
```

```

[-j rev1] [-j rev2] [-k kopt] modules...
-A      Reset any sticky tags/date/kopts.
-N      Don't shorten module paths if -d specified.
-P      Prune empty directories.
-R      Process directories recursively.
-c      "cat" the module database.
-f      Force a head revision match if tag/date not found.
-l      Local directory only, not recursive
-n      Do not run module program (if any).
-p      Check out files to standard output (avoids stickiness).
-s      Like -c, but include module status.
-r rev  Check out revision or tag. (implies -P) (is sticky)
-D date Check out revisions as of date. (implies -P) (is sticky)
-d dir  Check out into dir instead of module name.
-k kopt Use RCS kopt -k option on checkout. (is sticky)
-j rev  Merge in changes made between current revision and rev.
(Specify the --help global option for a list of other help options)

```

Vá para o diretório onde deseja trabalhar e crie uma cópia de trabalho com **checkout**.

```

Exemplo
mkdir /tmp/workdir
cd /tmp/workdir
cvs checkout exemplo-biblioteca-gnu
cd exemplo-biblioteca-gnu
ls -la

```

Observe que todos os arquivos do projeto foram copiados para o diretório `/tmp/workdir/exemplo-biblioteca-gnu`. Também foi criado o diretório `cvs`. Este diretório é mantido pelo programa `cvs`.

54.3.4 Para criar módulos

Bem, com o comando `checkout`, fizemos uma cópia de trabalho do projeto `exemplo-biblioteca-gnu`. Mas o nome `exemplo-biblioteca-gnu` é muito extenso e seria melhor um nome abreviado. Um módulo é exatamente isto, um nome abreviado para uma path grande no diretório do repositório. Veja a seguir como criar um módulo.

1. Baixa o arquivo `modules`, localizado em `/home/REPOSITORY/CVSRROOT/modules`

```
cvs checkout CVSRROOT/modules
```

2. Edita o arquivo `modules`

```
emacs CVSRROOT/modules
```

3. Inclua a linha abaixo (`nome_módulo path`)

```
lib-gnu exemplo-biblioteca-gnu
```

4. Salva o arquivo e envia para o repositório com o comando

```
cvs commit -m
    "adicionado o módulo exemplo-biblioteca-gnu ->lib-gnu"
```

O comando **commit** é usado para devolver para o repositório todos os arquivos novos ou modificados. Veja na listagem a seguir o protótipo do comando **commit**.

Listing 54.7: Saída do comando: `cvs -H commit`

```
[andre@mercurio cvs]$ cvs -H commit
Usage: cvs commit [-nRlf] [-m msg | -F logfile] [-r rev] files...
  -n          Do not run the module program (if any).
  -R          Process directories recursively.
  -l          Local directory only (not recursive).
  -f          Force the file to be committed; disables recursion.
  -F logfile  Read the log message from file.
  -m msg      Log message.
  -r rev      Commit to this branch or trunk revision.
(Specify the --help global option for a list of other help options)
```

Veja na listagem a seguir a saída do comando **commit** executada no diretório de trabalho após a modificação do arquivo `CVSRROOT/modules`.

Listing 54.8: Saída do comando `cvs commit` após adição de um módulo

```
[andre@mercurio workdir]$ cvs commit -m "adicionado o módulo exemplo-biblioteca
-gnu -> lib-gnu"
cvs commit: Examining CVSRROOT
cvs commit: Examining exemplo-biblioteca-gnu
cvs commit: Examining exemplo-biblioteca-gnu/.libs
Checking in CVSRROOT/modules;
/home/REPOSITORY/CVSRROOT/modules,v <-- modules
new revision: 1.2; previous revision: 1.1
done
cvs commit: Rebuilding administrative file database
```

Agora você pode executar o comando **checkout** de forma abreviada, usando o nome do módulo.

```
mkdir /tmp/workdir2
cd /tmp/workdir2
cvs checkout lib-gnu
```

Para que o comando ficasse ainda mais curto, poderia-se ter utilizado a forma abreviada de **checkout**.

```
cvs co lib-gnu
```

54.3.5 Para adicionar/remover arquivos e diretórios

O comando **add** agenda a adição de arquivos e diretórios que só serão copiados para o repositório com o comando **commit**. Da mesma forma, o comando **remove** agendam a remoção de arquivos e diretórios que só serão removidos do repositório com o comando **commit**.

Veja a seguir o protótipo destes comandos. Observe que para os comandos funcionarem, você deve estar no diretório de trabalho (`/tmp/workdir`).

Para adicionar um arquivo

Vamos criar um arquivo `leiametext`, o mesmo contém alguma informação sobre o projeto. Vamos criá-lo com o editor `emacs` (use o que lhe convier).

```
emacs leiametext
...inclue observações no arquivo leiametext...
```

Agora, vamos agendar a adição do arquivo com o comando `add`. A saída do comando é apresentada em itálico.

```
cvs add -m "adicionado arquivo leiametext" leiametext
cvs add: scheduling file 'leiametext' for addition
cvs add: use 'cvs commit' to add this file permanently
```

Depois de modificar outros arquivos, podemos efetivamente adicionar o arquivo `leiametext` no repositório usando o comando `commit`. Observe, em itálico, a saída gerada pelo comando `commit`.

```
cvs commit
cvs commit: Examining .
cvs commit: Examining .libs
cvs commit: Examining novoDir
RCS file:/home/REPOSITORY/emplo-biblioteca-gnu/leiametext,v done
Checking in leiametext;
/home/REPOSITORY/emplo-biblioteca-gnu/leiametext,v <--
leiametext initial revision: 1.1
done
```

Alguns comandos do programa `cvs` podem abrir um editor de texto para que você inclua alguma mensagem relativa a operação que foi realizada. No exemplo acima, depois do **`cvs commit`**, o `cvs` abriu o editor `emacs`. Na sua máquina provavelmente irá abrir o `vi`. Você pode alterar o editor a ser aberto pelo `cvs`, setando no arquivo `~/.bash_profile` a variável de ambiente `CVSEEDITOR` (Em minha máquina: `export CVSEEDITOR=emacs`).

Para adicionar vários arquivos:

O procedimento é o mesmo, primeiro agenda a adição com `add` e depois adiciona efetivamente com `commit`.

```
cvs add -m "adicionados diversos arquivos" *
cvs commit
```

Para adicionar um diretório:

A sequência envolve a criação do diretório (`mkdir novoDir`), o agendamento da adição (`cvs add novoDir`), e a efetiva adição do diretório com `commit`.

```
mkdir novoDir
cvs add novoDir
cvs commit -m "adicionado novo diretório" novoDir
```

Para adicionar toda uma estrutura de diretórios num projeto existente:

É o mesmo procedimento utilizado para importar todo um projeto. A única diferença é que a path de importação no repositório vai estar relativa a um projeto já existente. Veja o exemplo:

```
cd novoDir
cvs import -m 'msg' path_proj_no_repositorio/novodir
               nome_release nome_tag.
```

Para remover um arquivo:

Você deve remover o arquivo localmente, agendar a remoção e então efetivar a remoção com commit.

```
rm leiametext
cvs remove leiametext
cvs commit leiametext
```

O comando a seguir remove o arquivo localmente e no cvs ao mesmo tempo.

```
cvs remove -f leiametext
```

Para remover vários arquivos:

Você deve remover os arquivos, agendar a remoção e então remover efetivamente com commit.

```
rm -f *
cvs remove
cvs commit -m "removidos diversos arquivos"
```

Dica: Se você fizer alterações locais em um arquivo e depois remover o arquivo, não poderá recuperá-las. Para que possa recuperar as alterações, deve criar uma versão do arquivo usando o comando commit.

Para remover diretórios:

Vá para dentro do diretório que quer deletar, e delete todos os arquivos e o diretório usando:

```
cd nomeDir
cvs remove -f *
cvs commit
//A seguir delete o diretório:
cd ..
cvs remove nomeDir/
cvs commit
```

Para renomear arquivos:

Vá para dentro do diretório onde esta o arquivo a ser renomeado e execute os passos:

```

cd diretorio
mv nome_antigo nome_novo
cvs remove nome_antigo
cvs add nome_novo
cvs commit -m "Renomeado nome_antigo para nome_novo"

```

54.3.6 Para atualizar os arquivos locais

Como o cvs permite o trabalho em grupo. Um segundo usuário pode ter copiado e alterado os arquivos do projeto no repositório.

Um segundo usuário realizou as tarefas a seguir²:

```

mkdir /tmp/workdir3
cd /tmp/workdir3
cvs checkout lib-gnu
cd lib-gnu
emacs arquivo-usuario2.txt
cvs add arquivo-usuario2.txt
cvs commit -m "arquivo adicionado pelo usuario2"

```

Se outros usuários do projeto modificaram os arquivos do repositório, então os arquivos com os quais você está trabalhando podem estar desatualizados. Isto é, se um outro usuário modificou algum arquivo do repositório, você precisa atualizar os arquivos em seu diretório de trabalho.

Bastaria realizar um comando **cvs commit** devolvendo para o repositório todos os arquivos que você modificou, e um comando **cvs checkout**, que copiaria todos os arquivos do repositório, atualizados, para seu diretório de trabalho. Mas este procedimento pode ser lento. Seria mais rápido se o cvs copia-se para seu diretório de trabalho apenas os arquivos novos e modificados. É exatamente isto que o comando **update** faz. O protótipo do comando update é listado a seguir.

Listing 54.9: Saída do comando: cvs -H update

```

[andre@mercurio cvs]$ cvs -H update
Usage: cvs update [-APCdf1Rp] [-k kopt] [-r rev] [-D date] [-j rev]
      [-I ign] [-W spec] [files...]
      -A      Reset any sticky tags/date/kopts.
      -P      Prune empty directories.
      -C      Overwrite locally modified files with clean repository copies.
      -d      Build directories, like checkout does.
      -f      Force a head revision match if tag/date not found.
      -l      Local directory only, no recursion.
      -R      Process directories recursively.
      -p      Send updates to standard output (avoids stickiness).
      -k kopt Use RCS kopt -k option on checkout. (is sticky)
      -r rev  Update using specified revision/tag (is sticky).
      -D date Set date to update from (is sticky).
      -j rev  Merge in changes made between current revision and rev.
      -I ign  More files to ignore (! to reset).
      -W spec Wrappers specification line.

```

²Observe que o nome do diretório obtido pelo usuário 1 é exemplo-biblioteca-gnu e do usuário 2 lib-gnu. Isto é, se você usa *cvs checkout path_proj_no_repositorio* o cvs cria o diretório *path_proj_no_repositorio*. Se você usa *cvs checkout nome_modulo*, o cvs cria o diretório *nome_modulo*.

Veja no exemplo como deixar seu diretório de trabalho com os arquivos atualizados.

```
cd /tmp/workdir
cvs update
cvs update: Updating . U arquivo-usuario2.txt
cvs update: Updating .libs
cvs update: Updating novoDir
```

Observe que o arquivo “*arquivo-usuario2.txt*” criado pelo usuário 2 foi adicionado a sua cópia de trabalho.

54.4 Versões, tag's e releases

Descrevemos no início deste capítulo o que é um release e um tag. Apresenta-se a seguir como criar e usar releases e tags.

54.4.1 Entendendo as versões

Todos os arquivos do projeto que foram importados ou adicionados ao repositório tem uma versão. A versão é definida automaticamente pelo programa cvs e se aplica aos arquivos individualmente, isto é, cada arquivo tem sua versão.

De uma maneira geral a versão do arquivo é redefinida a cada alteração do arquivo que foi comutada com o repositório. Assim se o arquivo *leiametext*, que tem a versão 1.1, foi alterado. Quando o mesmo for devolvido ao repositório com o comando **cvs commit**, o mesmo passa a ter a versão 1.2. Veja Figura 54.1.

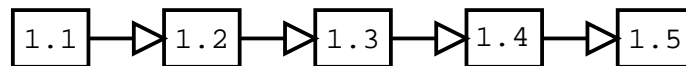


Figura 54.1: Versões de um arquivo.

No exemplo a seguir vai para o diretório de trabalho e modifica o arquivo *leiametext*. Depois realiza um commit.

```
cd /tmp/workdir/exemplo-biblioteca-gnu
emacs leiametext
...faça alterações no arquivo leiametext...e depois salve o arquivo.
cvs commit
cvs commit: Examining .
cvs commit: Examining .libs
cvs commit: Examining novoDir
Checking in leiametext;
/home/REPOSITORY/exemplo-biblioteca-gnu/leiametext,v <-- leiametext
new revision: 1.2; previous revision: 1.1 done
```


54.4.2 Para criar tag's

Como dito acima, cada arquivo do repositório vai ter uma versão. Entretanto, você pode realizar diversas modificações no arquivo `leiametext` (1.1 -> 1.2 -> 1.3 -> 1.4 -> 1.5), algumas modificações no arquivo `makefile` (1.1 -> 1.2 -> 1.3) e nenhuma modificação no arquivo `NomePrograma.cpp` (1.1). Ou seja, cada arquivo tem um número de versão diferente. Seria interessante se você pudesse se referir a todos os arquivos do projeto em uma determinada data com um mesmo nome simbólico. Um **tag** é exatamente isto, um nome simbólico usado para obter os arquivos do projeto em determinada data.

Veja na Figura 54.2 como é criado um novo tag. Observe que a versão de cada arquivo não é alterada.

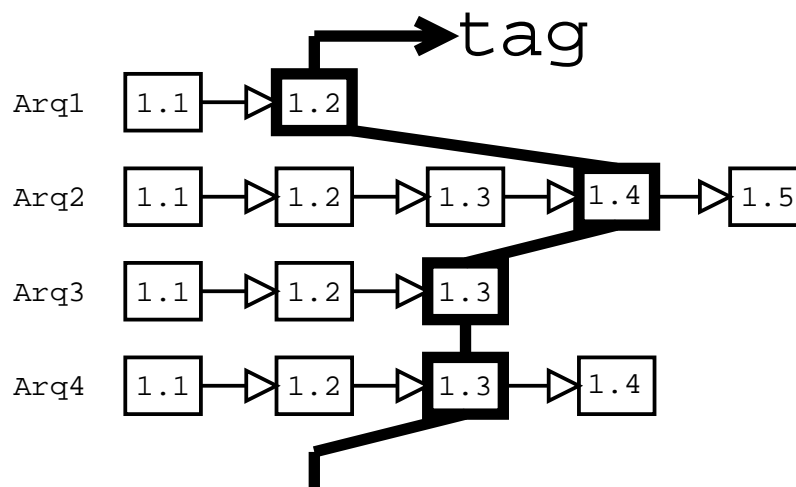


Figura 54.2: Criando um tag.

Assim, em determinado dia eu quero criar um tag simbólico, um nome que vou utilizar para todos os arquivos do projeto naquela data.

Protótipo para criar um tag para um único arquivo:

```
cd /tmp/workdir
cvs tag nome_release_simbolico nome_arquivo
```

Protótipo para criar um tag para todos os arquivos do projeto:

```
cd /tmp/workdir
cvs tag nome_release_simbolico
```

Veja na listagem a seguir a saída do comando, `cvs tag tag1` executada em nosso diretório de trabalho.

Listing 54.10: Saída do comando: `cvs -tag nome`

```
[andre@mercurio exemplo-biblioteca-gnu]$ cvs tag tag1 *
cvs tag: warning: directory CVS specified in argument
cvs tag: but CVS uses CVS for its own purposes; skipping CVS directory
T arquivo-usuario2.txt
T doxygem.config
```

```
T e87-Polimorfismo
T e87-Polimorfismo.cpp
T e87-PolimorfismoDinamic.cpp
T e87-PolimorfismoStatic.cpp
T e87-Programa.cpp
T e87-TCirculo.cpp
T e87-TCirculo.h
T e87-TElipse.cpp
T e87-TElipse.h
T e87-TPonto.cpp
T e87-TPonto.h
T leiame.txt
T makefile
T Makefile
T makefile-libtool
T uso-makefile
cvs tag: Tagging novoDir
```

Para recuperar a versão completa do projeto usando o tag que acabamos de criar:

```
cd /tmp/workdir/exemplo-biblioteca-gnu
cvs checkout -r tag1 lib-gnu
```

Observe que para baixar o módulo lib-gnu usamos **cvs checkout lib-gnu**, e para baixar o tag1 do módulo lib-gnu, usamos, **cvs checkout -r tag1 lib-gnu**. Ou seja, apenas adicionamos após o comando checkout, o parâmetro -r e o nome do tag.

54.4.3 Para criar release's

Geralmente utilizamos um tag para criar uma versão do projeto que esteja funcionando, ou que compreenda a finalização de um determinado conjunto de tarefas que estavam pendentes. Assim, com o nome do tag você pode recuperar o projeto naquela data usando um nome abreviado.

Entretanto, depois de finalizado o programa ou uma versão funcional, você pode criar um release do programa. A diferença entre o tag e o release, é que o tag não modifica a versão dos arquivos do projeto. O release modifica a versão de todos os arquivos, dando a todos os arquivos um mesmo número de versão.

- Um release é geralmente um pacote funcional, se aplica a todos os arquivos do projeto.
- Depois de definido o release o mesmo não pode ser modificado.
- Você deve criar um release sempre que tiver finalizado uma parte importante de seu programa.

Veja Figura 54.3 como fica um novo release.

Veja a seguir o protótipo para criar um release.

Protótipo:

```
cvs commit -r número__release

cd /tmp/workdir
cvs commit -r 2
```

Além de criar o release, abre o vi³, para edição de um arquivo de log. Inclua algum comentário a

³ou o editor setado com CVSEDITOR. No vi digite esc :q para sair, esc :q!. para sair sem salvar alterações.

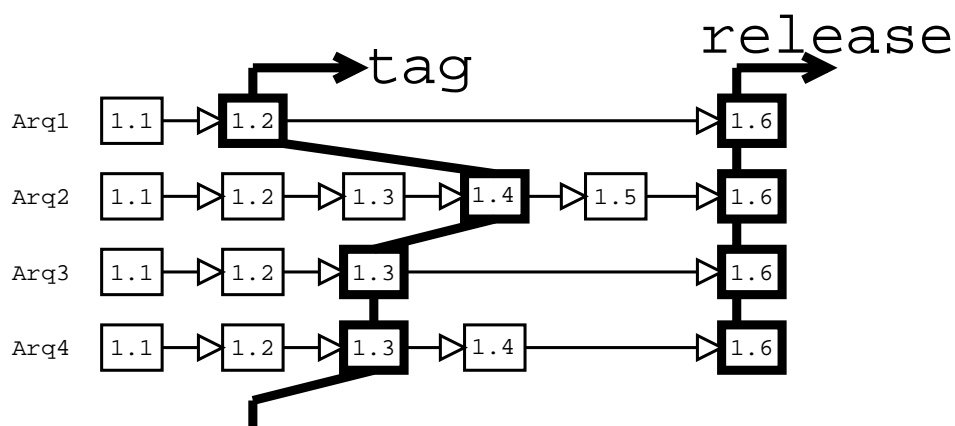


Figura 54.3: Criando um release.

respeito do release que foi criado.

Veja na listagem a seguir a saída do comando `cvcs commit -r 2`.

Listing 54.11: Saída do comando: `cvcs commit -r 2`

```
[root@mercurio lib-gnu]# cvcs commit -r 2
cvcs commit: Examining .
cvcs commit: Examining .libs
cvcs commit: Examining novoDir
Checking in Makefile;
/home/REPOSITORY/exemplo-biblioteca-gnu/Makefile,v <-- Makefile
new revision: 2.1; previous revision: 1.1
done
Checking in arquivo-usuario2.txt;
/home/REPOSITORY/exemplo-biblioteca-gnu/arquivo-usuario2.txt,v <-- arquivo-
usuario2.txt
new revision: 2.1; previous revision: 1.1
done
....
....
Checking in leiname.txt;
/home/REPOSITORY/exemplo-biblioteca-gnu/leiname.txt,v <-- leiname.txt
new revision: 2.1; previous revision: 1.3
done
```

Protótipo para criar um release e já deletar a cópia do diretório local:

```
cvcs release -d diretório_de_trabalho
```

54.4.4 Recuperando módulos e arquivos

O `cvcs` permite que tanto os códigos novos como os antigos possam ser recuperados. De uma maneira geral basta passar o nome do arquivo e sua versão (tag, release, módulo).

Protótipo para recuperar um release:

```
#Pode-se baixar um release antigo, passando o nome do release.
```

```
cvs checkout -r nome_release path_projeto_no_cvs
#ou o nome do módulo
cvs checkout -r nome_release nome_modulo
```

Protótipo para recuperar um arquivo de uma versão antiga:

```
cvs update -p -r nome_release nome_arquivo > nome_arquivo
```

-p Envia atualizações para saída padrão (a tela).
 -r nome_release Indica a seguir o nome do release.
 nome_arquivo O nome do arquivo a ser baixado
 > nome_arquivo Redireciona da tela para o arquivo nome_arquivo.

No exemplo a seguir, recupera o arquivo leiname.txt do tag1.

```
cvs update -p -r tag1 leiname.txt > leiname-tag1.txt
=====
Checking out leiname.txt RCS:
/home/REPOSITORY/exemplo-biblioteca-gnu/leiname.txt,v VERS: 1.2
*****
```

54.5 Para verificar diferenças entre arquivos

O programa cvs tem suporte interno ao programa diff (apresentado no Capítulo 43), permitindo comparar os arquivos que estão sendo usados localmente com os do repositório.

Protótipo:

```
#Compara arq local e arq do repositório
cvs diff arq
#Verifica diferenças de todos os arquivos
cvs diff
```

O usuário 2, modificou o arquivo leiname.txt depois de criado o release 2. Veja na listagem a seguir a saída do comando cvs diff, executado pelo usuário 1.

Listing 54.12: Saída do comando: cvs-diff

```
[andre@mercurio exemplo-biblioteca-gnu]$ cvs diff
cvs diff: Diffing .
Index: leiname.txt
=====
RCS file: /home/REPOSITORY/exemplo-biblioteca-gnu/leiname.txt,v
retrieving revision 2.2
diff -r2.2 leiname.txt
7,11d6
< Alteração realizada depois de criado o tag1.
<
<
< Modificações realizadas depois do release.
< Pelo usuário 2.
cvs diff: Diffing .libs
cvs diff: Diffing novoDir
```

54.6 Verificando o estado do repositório

O cvs tem um conjunto de comandos que você pode usar para verificar o estado dos arquivos armazenados no repositório.

54.6.1 Histórico das alterações

Você pode obter uma lista com o histórico das alterações realizadas.

Mostra: data, hora, usuário, path usada (ou módulo, ou ramo), diretório de trabalho:

Protótipo:

```
cvs history
```

54.6.2 Mensagens de log

Você pode obter uma lista dos log's do arquivo.

Mostra: path no repositório, versão, nomes simbólicos, revisões e anotações realizadas.

Protótipo:

```
cvs log arquivo
```

Veja a seguir a saída do comando **cvs -log leiname.txt**. Observe as diferentes revisões e anotações, o nome do autor. Observe os nomes simbólicos.

Listing 54.13: Saída do comando: cvs -log leiname.txt

```
RCS file: /home/REPOSITORY/exemplo-biblioteca-gnu/leiname.txt,v
Working file: leiname.txt
head: 2.2
branch:
locks: strict
access list:
symbolic names:
    tag1: 1.2
keyword substitution: kv
total revisions: 5;      selected revisions: 5
description:
adicionado arquivo leiname.txt
-----
revision 2.2
date: 2002/08/12 23:28:55;  author: andre;  state: Exp;  lines: +4 -0
Modificações realizadas no leiname.txt depois de criado o release.
-----
revision 2.1
date: 2002/08/12 23:12:05;  author: andre;  state: Exp;  lines: +0 -0
Criado o release 2.
-----
revision 1.3
date: 2002/08/12 23:10:32;  author: andre;  state: Exp;  lines: +1 -0
Alterações no leiname.txt depois de criado o tag1.
-----
revision 1.2
date: 2002/08/12 22:45:56;  author: andre;  state: Exp;  lines: +5 -0
```

```

Modificações no arquivo leiname.txt
-----
revision 1.1
date: 2002/08/12 21:33:43; author: andre; state: Exp;
Efetivamente adicionado o arquivo leiname.txt
=====

```

54.6.3 Anotações

Você pode obter uma lista das anotações realizadas.

Mostra: versão, nome usuário, data, mensagem.

Protótipo:

cvs annotate

54.6.4 Verificando o status dos arquivos

O comando status mostra uma série de informações a respeito do arquivo. O mesmo pode ser utilizado para verificar quais arquivos precisam ser atualizados. Veja a seguir o protótipo.

Protótipo:

cvs status

-v *Mostra ainda os tag's.*
-R *Processamento recursivo.*
-l *Somente este diretório.*

Informações listadas pelo comando status:

Up-to-date O arquivo não foi alterado.

Locally modified O arquivo foi modificado localmente.

Locally added O arquivo foi adicionado localmente.

Locally removed O arquivo foi removido localmente.

Needs checkout O arquivo foi alterado por terceiro e precisa ser atualizado (Com um update baixa o arquivo mesclando-o com o local. Com um commit atualiza no servidor).

File had conflicts on merge O arquivo apresenta conflitos após a mistura.

Veja na listagem a seguir a saída do comando status. Observe que o arquivo foi localmente modificado.

Listing 54.14: Saída do comando: `cvs -status leiname.txt`

```
[andre@mercurio exemplo-biblioteca-gnu]$ cvs status leiname.txt
=====
File: leiname.txt          Status: Locally Modified

Working revision:   2.2      Result of merge
Repository revision: 2.2      /home/REPOSITORY/exemplo-biblioteca-gnu/leiname.
txt,v
Sticky Tag:         (none)
Sticky Date:        (none)
Sticky Options:     (none)
```

54.7 Ramos e Misturas (Branching and Merging)

O programa `cvs` permite que você crie um ramo principal para seu projeto e ramos derivados. Posteriormente você pode misturar os diferentes ramos.

Veja na Figura 54.4 a disposição de um novo ramo.

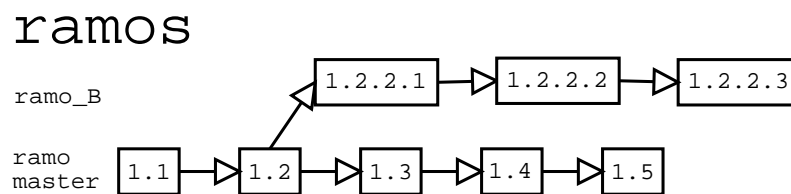


Figura 54.4: Como ficam os ramos.

Depois de finalizado um release de um programa, é bastante usual a criação de três ramos.

Digamos que você esteja trabalhando do projeto `gnome`, e que o release 1.0 já foi suficientemente testado, podendo ser publicado. Então, você cria o release 1.0.

Release `gnome 1.0`.

Observe que é a versão final do `gnome 1.0`.

Agora você pode criar um ramo de `patch`, o mesmo vai conter os arquivos da versão 1.0, mas com correções de bugs que tenham sido localizados. Assim, se foi identificado algum bug na versão 1.0, você faz as alterações no ramo `gnome 1.0-patch`, deixando o release 1.0 inalterado.

Ramo: `gnome 1.0-patch`

Você pode criar um ramo novo, onde ficarão os arquivos da nova versão do `gnome`.

Ramo: `gnome 1.1`

Ou seja, vamos ter três ramos. O release 1.0 que não será mais alterado. O `patch` que vai ter as correções de bugs da versão 1.0 e o 1.1 que terá os arquivos da nova geração do `gnome`.

54.7.1 Trabalhando com ramos

Para criar um ramo a partir da cópia de trabalho local (-b de branch):

```
cv$ tag -b nome_do_amo
```

Para criar um ramo a partir de um release existente, sem uma cópia de trabalho local:

```
cv$ rtag -b -r nome_do_release nome_do_amo path_no_repositorio
```

Baixando um ramo:

```
cv$ checkout -r nome_do_amo path_no_repositorio
```

Atualização dos arquivos locais de um dado ramo:

```
cv$ update -r nome_do_amo path_no_repositorio
```

ou

```
cv$ update -r nome_do_amo nome_modulo
```

Para saber com qual ramo você está trabalhando, verifique o nome do ramo em “Existing tags”.

```
cv$ status -v nome_arquivo
```

54.7.2 Mesclando 2 versões de um arquivo

Com a opção -j, você pode verificar as diferenças entre 2 versões de um arquivo. Veja o protótipo e um exemplo a seguir.

Protótipo:

```
cv$ update -j versãoNova -j versãoVelha nomeArquivo
```

```
cv$ update -j 2 -j tag1 leiname.txt
U leiname.txt
RCS file: /home/REPOSITORY/exemplo-biblioteca-gnu/leiname.txt,v
retrieving revision 2.2
retrieving revision 1.2
Merging differences between 2.2 and 1.2 into leiname.txt
```

Observe a mensagem apresentada. O cvs recupera a versão 2.2 (relativa ao release -j 2) e a versão 1.2 (relativa ao tag1) e mistura as duas no arquivo leiname.txt.

54.7.3 Mesclando o ramo de trabalho com o ramo principal

Digamos que você está trabalhando no ramo principal. Que um segundo usuário criou o ramo `_B` e fez alterações no ramo `_B`. Agora você quer incluir as alterações do ramo `_B` no ramo principal.

1. Baixa o módulo de trabalho

```
cv$ checkout nome_modulo
```

2. Baixa o upgrade do ramo `_B`. Ou seja, atualiza os arquivos locais mesclando os mesmos com os do ramo `_B`.

```
cv$ update -j ramo_B
```

3. Resolve os possíveis conflitos. Alguns arquivos que tenham sido modificados por outros usuários podem ter conflitos de código, você precisa resolver estes conflitos.

Correção de possíveis conflitos de código...

4. Copia os arquivos de volta para o repositório, atualizando o repositório.

```
cv$ commit -m "Ramo mestre mesclado com ramo_B"
```

5. Para deletar o diretório local de trabalho.

```
rm -f -r path_local/
```

54.8 Configuração do cvs no sistema cliente-servidor

Neste tipo de configuração o projeto principal fica na máquina servidora (ou seja o repositório fica no servidor). O usuário baixa o programa para sua máquina local usando checkout, faz modificações e depois copia as modificações para o repositório usando o comando commit.

- O servidor para uso do cvs pode ser um micro pouco potente (133MHz, 32Mb), com HD suficiente (4 vezes o tamanho do projeto).
- O acesso ao repositório é dado por:
:tipo_de_acesso:path_do_projeto

onde tipo_de_acesso:

:local: Você está na máquina servidora: Se estiver trabalhando na mesma máquina do repositório, você faz um acesso local ao projeto e pode acessar os arquivos do projeto diretamente com **cv\$ checkout path_no_repositorio**.

:servidor: Você está na máquina cliente: Se estiver remoto, deve-se incluir o nome do servidor

: **servidor: user@hostname:/path/to/repository**

Ex:

```
export CVSROOT=:pserver: usuario1@nome_servidor:/path_repositorio
cv$ checkout path_no_repositorio.
```

- Consulte o manual do cvs para ver como configurar o servidor.

Exemplo:

Por default, a conexão do cvs usa o protocolo RSH. Assim, se andre esta na máquina mercurio.lmpt.ufsc.br e o servidor é enterprise.lmpt.ufsc.br no arquivo '.rhosts' deve ter a linha:

```
mercurio.lmpt.ufsc.br andre
```

Para testar:

```
rsh -l bach enterprise.lmpt.ufsc.br 'echo $PATH'
```

Deve-se setar na máquina cliente o endereço do programa cvs no servidor com a variável de ambiente CVS_SERVER.

54.8.1 Variáveis de ambiente

Variáveis de ambiente do cvs definidas no arquivo profile:

\$CVSROOT Diretório de trabalho do cvs.

\$CVS_SERVER Endereço do programa cvs na máquina servidora.

\$CVSEDITOR Editor default do cvs.

\$CVSUMASK Define o formato dos arquivos novos a serem criados.

54.9 Frontends (cervisia)

Existem front-ends para o programa cvs, de uma olhada no cervisia, o mesmo é encontrado no site

(<http://cervisia.sourceforge.net/>).

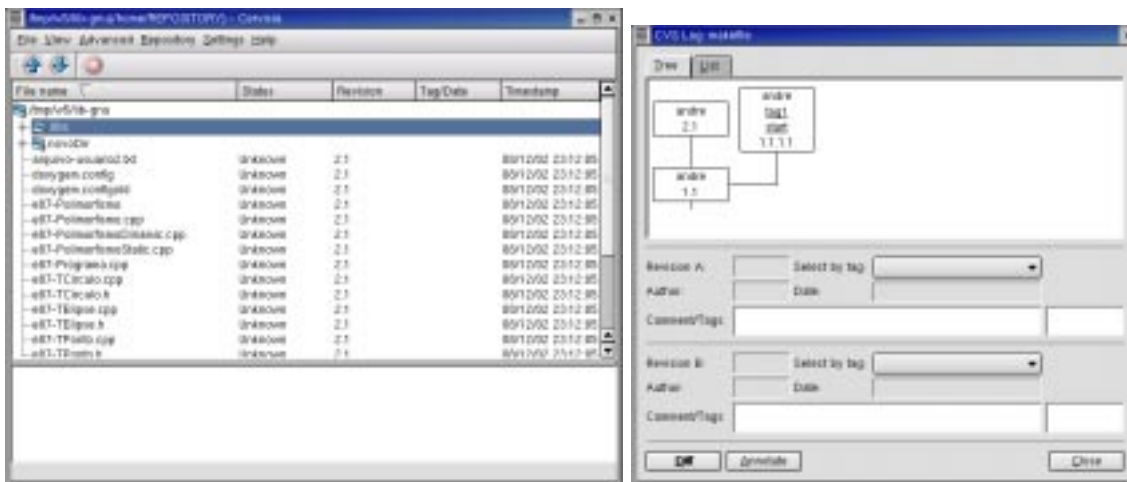


Figura 54.5: Um frontend para o cvs (o cervisia).

Dica: Você pode encontrar e baixar um refcard com os comandos do cvs. Use seu browser e mecanismo de pesquisa e procure por cvs refcard.

54.10 Sentenças para o cvs

- Quando você passa como parâmetro de algum comando do cvs um diretório. Todos os arquivos do diretório e subdiretórios sofrem o efeito do comando.
- Uma path, um módulo, um ramo são equivalentes. Um tag, um release, uma versão são equivalentes. Ou seja, se o programa cvs espera uma path_do_repositório você também pode passar o nome de um módulo ou de um ramo. Se o cvs espera um nome de versão, você pode passar o nome do tag, ou o nome do release.
- Monte um grupo de trabalho:
Para trabalhar em grupo em um projeto, você deve-se definir um grupo no Linux/Unix/Mac OS X. O grupo de trabalho terá acesso aos arquivos do repositório num sistema cliente-servidor.
- Pode-se definir diferentes formas de acesso aos arquivos, autenticações e sistemas de segurança. Dê uma olhada no manual de configuração do cvs.

54.11 Um diagrama com os comandos do cvs

Veja na Figura um diagrama com os comandos do cvs.

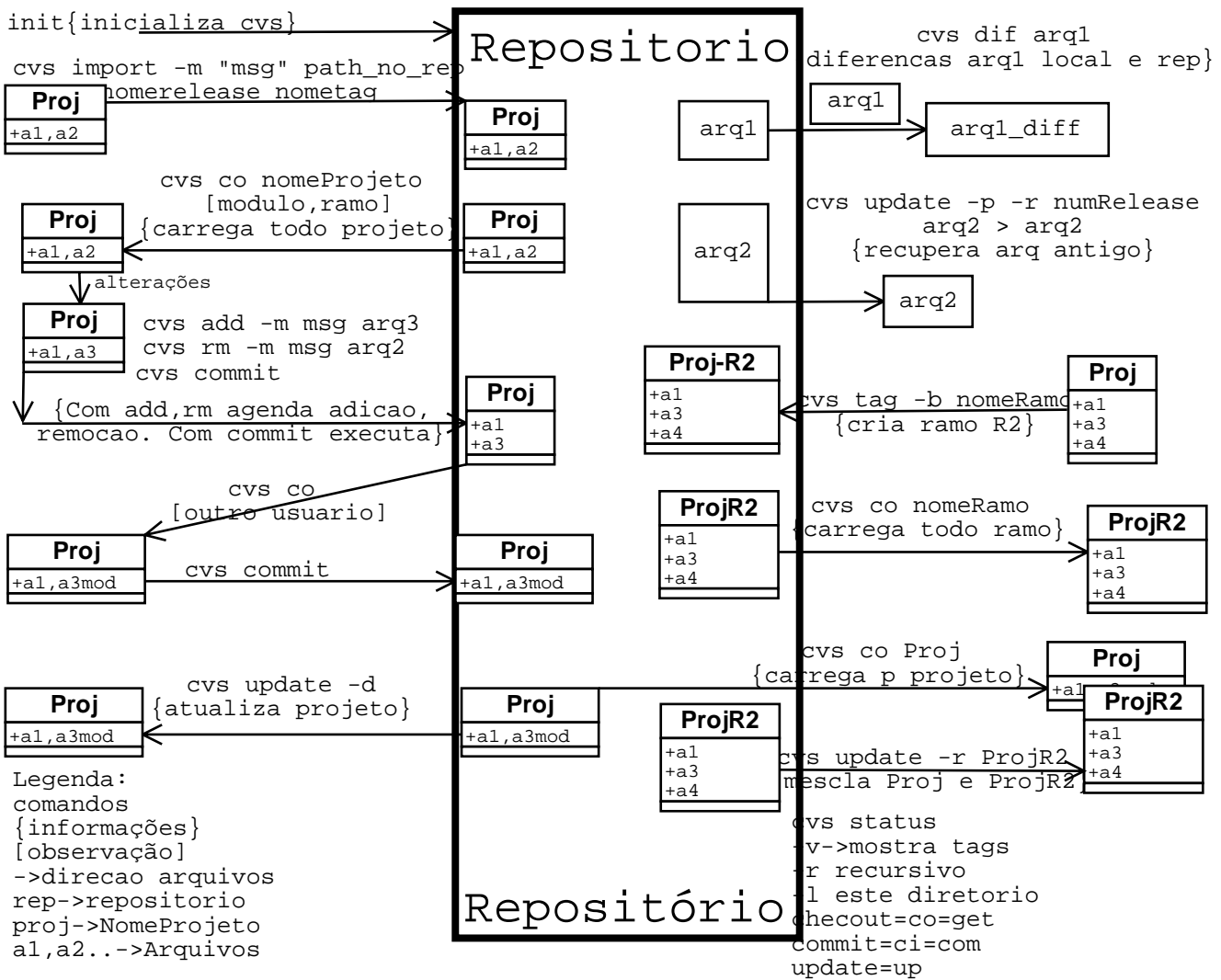


Figura 54.6: Diagrama com os comandos do cvs.

Parte VI

Modelagem Orientada a Objeto

Capítulo 55

Modelagem TMO (UML)

A modelagem orientada a objeto foi desenvolvida e aperfeiçoada por autores como Coad, Yourdan, Boock, Rumbaugh, Blaha, Premerlani, Eddy e Lorensen. Cada um destes autores desenvolveu uma metodologia diferente. Atualmente estas metodologias foram unificadas e foi desenvolvida a UML (Universal Modelling Language).

As vantagens do desenvolvimento de modelos podem ser resumidas a uma maior facilidade de testar uma entidade física antes de lhe dar forma final (modelos físicos), a maior facilidade na comunicação entre as diversas pessoas envolvidas (pelo uso de notação uniforme), a facilidade de visualização e a redução da complexidade dos sistemas.

Neste capítulo descreve-se a TMO. A metodologia TMO é descrita no livro "Modelagem e Projetos Baseados em Objetos" de Rumbaugh, Blaha, Premerlani, Eddy e Lorensen (1991), [?]. A TMO é muito semelhante a UML.

A metodologia TMO é dividida em três modelos, o modelo de objetos, o modelo dinâmico e o modelo. Os três modelos são ortogonais, ou seja se complementam, e a maior importância de um em relação ao outro vai depender do tipo de programa desenvolvido e do enfoque desejado em determinado momento. Os três modelos são descritos a seguir.

55.1 Modelo de objetos

O modelo de objetos é usado na descrição estática dos objetos e de seus relacionamentos. É um diagrama de objetos, onde constam as classes, seus atributos, métodos e os relacionamentos entre as diversas classes, como heranças (generalização/especialização), associações, agregações (todo parte).

Na maioria dos programas é o modelo mais importante, pois representa e identifica os objetos que compõem o sistema, além dos relacionamentos entre estes objetos.

Como é constituído de objetos, atributos e métodos que são facilmente reconhecidos tanto pelos clientes como pelos desenvolvedores, permite uma maior comunicação no desenvolvimento do programas.

Apresenta-se a seguir a definição, as nomenclaturas e conceitos usados no modelo de objetos.

55.1.1 Modelo de objetos->Ligações

Uma **ligação** é uma conexão física ou conceitual entre objetos.

Exemplo:

- No relacionamento de dois objetos podemos ter o surgimento de uma propriedade que não pertence a nenhum dos objetos originais, só existe quanto ocorre a interação.
- Na interação do objeto matriz sólida com o objeto fluido, surge a propriedade tensão interfacial. A tensão interfacial é uma propriedade de ligação.

Uma ligação pode ser representada através de uma classe, e neste sentido pode ter seus próprios atributos, denominados de **atributos de ligação**. Incluir os atributos de uma ligação dentro de uma das duas classes existentes reduz a possibilidade de modificações no projeto, o projeto fica mais restrito.

55.1.2 Modelo de objetos->Associações

Uma **associação** é uma ligação conceitual entre classes, geralmente aparecem como verbos nas especificações do programa e são intrinsecamente bidirecionais. Geralmente, as associações são implementadas com o uso de ponteiros, mas não devem ser representadas no modelo de objetos com ponteiros, porque uma associação só existe se existirem os dois objetos, e não fazer parte de nenhum dos dois objetos.

As associações podem ser ternárias ou de ordem mais elevada, mas na prática a maioria são binárias.

Exemplo:

- Um jogador de futebol e uma bola, são dois objetos independentes, mas se desejamos um jogo de futebol teremos estes dois objetos se relacionando.
- Quando um gerente solicita a um funcionário a realização de uma determinada tarefa.

Cardinalidade: A cardinalidade (ou multiplicidade) indica através de números as quantidades de cada objetos em uma associação. Quando você tem uma cardinalidade maior que um você vai ter um grupo de objetos se relacionando através de uma associação. Quando este grupo de objetos for **ordenado** você os representa através da palavra ordenado entre colchetes {ordenado}.

Papel²: Um papel é uma extremidade de uma associação. Uma associação binária tem dois papéis. Um papel deve ter um nome unívoco, que identifique com clareza a extremidade da associação. Como um nome de papel pode representar um atributo do objeto, ele não deve ter o mesmo nome dos atributos do objeto.

Qualificador²: Uma qualificação inter-relaciona classes de objetos, serve para distinguir um objeto na extremidade muitos. É usada na representação das instâncias da classe.

Dicionário²: Se o projeto for muito grande pode-se criar uma lista de nomes de classes, atributos e métodos, com o objetivo de evitar repetições e confusões.

Dicas²: Evitar associações ternárias, não amontoar atributos de ligação em uma classe. Usar associações qualificadas onde for possível, evitar associações 1:1, geralmente ela é uma associação 1:0 ou 1:n, significando que o objeto associado pode ou não existir.

55.1.3 Modelo de objetos->Agregação

Uma agregação representa uma associação específica, em que um dos objetos é usado para formar o outro. Diz-se "**todo-parte**" em que *todo* representa o objeto composto e *parte* uma das partes que o compõem. Pode dizer ainda "**uma parte de**", "**tem um**".

Quando unimos vários objetos simples para criar um objeto mais complexo, estamos utilizando uma estrutura todo-parte.

A propriedade mais significativa da agregação é a transitividade, se A é parte de B e B é parte de C, então A é parte de C. É ainda anti-simétrica, ou seja se A é parte de B, B faz parte de A. Em alguns casos os dois objetos só podem ter existência juntos.

Exemplo:

- Como exemplo, um livro e sua capa, não tem muito sentido falar de um livro sem capa.

Uma **agregação** pode ser **recursiva**, isto significa que um objeto pode conter um objeto de seu tipo.

Exemplo:

- Um filme que fala de outro filme.

Se houver dúvida se um objeto é ou não uma agregação de outro deve-se usar uma associação. Veja abaixo como identificar se uma associação é ou não uma agregação.

Será uma agregação se:

Se os dois objetos forem formados por um relacionamento todo parte.

- Você usar a expressão parte de.
- As operações executadas sobre um forem também executadas sobre o outro.
- As operações executadas em um se propagam para o outro.

Será uma associação se:

- Se os dois objetos forem normalmente considerados separados.

Propagação: Ocorre quando uma operação aplicada sobre um objeto se estende para os demais. Uma propagação é indicada por uma seta.

55.1.4 Modelo de objetos->Generalização e Herança

Através de uma generalização, pode-se montar códigos altamente reaproveitáveis. A abstração usando os conceitos de herança é poderosa, permitindo destacar o que é comum aos diversos objetos que compõem a herança, sem deixar de levar em conta as suas particularidades. Deve-se destacar ainda uma simplificação conceitual na análise do problema.

Generalização é a criação de classes descendentes (filhas) a partir de uma **classe pai**. As **classes filhas** tem acesso a todos os métodos e atributos da classe pai. Uma generalização pode ter um nome como "**é um tipo de**", ou "**é um**".

Assinatura, é o nome da função, seus parâmetros e tipo de retorno. Para a implementação do polimorfismo as funções devem ter a mesma assinatura.

Delegação²: é o mecanismo pelo qual um objeto transfere a execução de uma operação para outro. Neste caso a classe deve ser desenhada como herdeira da classe pai e deve possuir objetos das demais classes, delegando a estas algumas tarefas. Pode-se usar uma mistura de herança múltipla com elementos delegados. Pode-se ainda desenvolver uma herança multinivelada, neste caso os métodos e atributos são repetidos em classes no mesmo nível.

Discriminador²: Um discriminador é um atributo de enumeração que indica qual propriedade de um objeto está sendo abstraída por um determinado relacionamento de generalização.

Algumas regras para solucionar problemas de herança múltipla²:

- Se uma subclasse possui várias superclasses, mas de mesma importância, tente usar delegação.
- Se uma das classes pai é dominante, use herança desta classe e delegação para as demais.
- Se uma das classes pai é o gargalo do desempenho, esta deve ser a superclasse.
- Se for usar herança multinivelada, decomponha primeiro as classes mais importantes. Se entretanto as funções forem muitos grandes, a herança multinivelada deve ser evitada, pois o código das funções deve ser copiado.

55.1.5 Modelo de objetos->Módulo / Assunto

Um **módulo** ou **assunto** é um agrupamento de classes que se relacionam através de algum conceito. Classes que tem um mesmo comportamento básico.

Os nomes de classes e de associações devem ser únicos em um módulo/assunto, e devem ter um formato semelhante.

Folha: Uma **folha** é uma página impressa. Cada módulo (assunto) é composto por uma ou mais folhas. Deve-se evitar colocar mais de um assunto em uma folha.

Restrições: Em um modelo de objetos, podemos ter **restrições**, que estabelecem relações funcionais entre objetos, classes, atributos e ligações. Como exemplo podemos citar a restrição de que o raio de um círculo não ultrapasse determinado valor, a restrição de só poder desenhar dentro da área cliente de uma janela. Um modelo de objetos bem desenvolvido deve ter muitas restrições.

Sugestões práticas:

É preciso compreender o problema, mantê-lo simples, escolher nomes com cuidado, definir uma nomenclatura padrão. Faça com que outras pessoas revisem o seu modelo. Documente sempre o que fizer.

55.2 Modelo dinâmico

O modelo dinâmico é usado na descrição das transformações do objeto com o tempo. Formado por diagramas de estado que servem para o controle da execução do programa. O modelo dinâmico se preocupa com o controle da execução e não com os detalhes da execução. Representa a evolução da execução do programa, as resposta do programa aos eventos gerados pelo usuário.

Todo programa que envolva interação e tempo, ou ainda interface com o usuário e controle de processos, deve ter um bom modelo dinâmico.

55.2.1 Modelo dinâmico->Eventos²

Um evento pode ser um estímulo externo provocado pelo usuário, e que provoca a modificação do estado de um ou mais objetos. Um evento representa uma ação que ocorre em determinado tempo e tem duração zero.

Um evento pode ser provocado pelo usuário por exemplo ao pressionar o mouse, ou selecionar um item de menu, também pode ser provocado por um outro programa, ou pelo sistema operacional.

Uma chamada a uma sub-rotina não é um evento, pois retorna um valor.

Todo evento ocorre em determinado momento, a hora em que determinado evento ocorre é um atributo do evento.

Cada evento é uma ocorrência única, mas os eventos podem ser agrupados em grupos com propriedades comuns.

Os eventos simples não transportam atributos, mas a maioria transporta algum tipo de parâmetro.

Se dois eventos não são relacionados de forma causal eles são chamados **concorrentes**, um não tem efeito sobre o outro, não existe uma relação temporal entre eles (ordem de execução).

Um evento pode ser enviado de um objeto para outro, transferindo informações (unidirecional).

Nossa análise é que transforma um determinado evento em um evento de erro, ou seja nós é que interpretamos o evento como sendo um erro.

Ação²: *é uma operação instantânea e esta associada a um evento. As ações podem ainda representar operações internas de controle. A notação para uma transição é uma barra ("/") e o nome (ou descrição) da ação, em seguida ao nome do evento que a ocasiona.*

Ações de entrada e saída²: Quando as transições de um estado executam a mesma ação, pode-se vincular o estado a ação.

Ações internas (faça)²: Ações internas são escritas dentro do estado após uma barra invertida, e representam ações que não mudam o estado do objeto.

Envio de eventos²: Um evento pode se dirigir a um objeto ou a um conjunto de objetos.

55.2.2 Modelo dinâmico->Cenários

Um cenário representa uma determinada seqüência de eventos que ocorre na execução do programa.

Exemplo:

- Uma seqüência em que o usuário abre um arquivo do disco, realiza determinadas modificações no mesmo e salva o arquivo.

55.2.3 Modelo dinâmico->Estados

Os valores assumidos pelos atributos de um objeto representam o seu estado. Os estados representam intervalos de tempo.

Um diagrama de estado representa os diversos estados que um determinado objeto assume, é formado por nós que representam os estados e setas que representam as transições entre estados (eventos).

Um estado pode sofrer alterações qualitativas e quantitativas. Uma alteração quantitativa representa qualquer alteração em qualquer dos atributos do objeto. Por exemplo a alteração da temperatura da água de 55° para 56° Célsius. Já uma alteração qualitativa representa uma alteração conceitual do objeto, como a alteração da temperatura da água de 55° (líquida) para -5° (sólida). No exemplo acima o estado depende de uma condição (a temperatura da água).

Um estado depende dos eventos anteriores. Mas de uma forma geral os eventos anteriores são ocultados pelos posteriores.

Por uma questão de economia, em um estado só devem ser listados os atributos de interesse. O diagrama de estado da classe pai deve ser separado do da classe filha.

Um estado complexo pode ser dividido em diagramas de nível inferior.

Exemplo:

- O objeto água pode estar no estado sólido, líquido ou gasoso.

55.2.4 Modelo dinâmico->Diagrama de Estados²

O diagrama de estados é usado para descrever os diversos estados assumidos pelos objetos e os eventos que ocorrem.

Assim um estado recebe um evento e envia um evento. Um evento modifica um estado e esta modificação é chamada de **transição**.

Um diagrama de estado deve ser desenhado para cada classe de objetos; Assim cada instância de objeto compartilha o mesmo diagrama de estado, embora cada objeto tenha o seu estado.

Um diagrama de estado pode representar ciclos de vida, no caso em que um objeto é criado realiza determinados procedimentos e é eliminado. Pode representar laços contínuos, quando o objeto esta sempre vivo.

Condições: Uma condição impõe algum tipo de restrição aos objetos. Uma condição é representada entre colchetes `[]`.

Exemplo:

- Um determinado evento só será realizado se determinadas condições forem satisfeitas.

Atividade²: é uma operação que consome determinado tempo para ser executada e esta associada a um estado. Dentro de um retângulo com o nome do estado pode existir uma instrução "**faça: X**", informando que a atividade X vai ser executada.

Diagrama de estados nivelados²: Pode-se representar mais detalhadamente um estado, usando-se diagramas de estados nivelados. Cada estado do diagrama nivelado representa um sub-estado, semelhante a sub-rotinas. Um rótulo no olho de boi indica o evento gerado no diagrama de estados de alto nível.

Concorrência e Sincronização no interior de um objeto²: Uma concorrência dentro de um estado subdivide o estado em dois. A notação é uma seta que se divide em uma linha pontilhada grossa. Posteriormente as duas atividades podem convergir, a notação é a união das setas.

Transição automática²: Uma seta sem um nome de evento indica uma transição automática que é disparada quando a atividade associada com o estado de origem esta completa.

55.2.5 Sugestões práticas

- Só construa diagramas de estados para classes de objetos com comportamento dinâmico significativo. Considere apenas atributos relevantes ao definir um estado.
- Verifique a consistência dos diversos diagramas de estados relativamente aos eventos compartilhados para que o modelo dinâmico completo fique correto.
- Use cenários para ajudá-lo.
- Os diagramas de estados das subclasses devem concentrar-se em atributos pertencentes unicamente as sub-classes.
- Os diagramas de estados dos diversos objetos se combinam para formar o diagrama dinâmico. As ligações entre os diversos diagramas é realizada pelas mensagens (eventos) compartilhados.

55.2.6 Relacionamento do modelo dinâmico com o modelo de objetos

A estrutura de um modelo dinâmico é estreitamente relacionada com o modelo de objetos desenvolvida. Os eventos podem ser representados como operações no modelo de objetos. A hierarquia de estados de um objeto é equivalente a um conjunto de restrições do modelo de objetos. Os eventos são mais expressivos que as operações, porque o efeito de um evento não depende somente da classe do objeto, mas também de seu estado.

55.3 Modelo funcional³

O último modelo, o funcional, é usado para uma descrição da interação dos diversos métodos das classes, descreve o fluxo de dados entre objetos. Os diagramas desenvolvidos contam com nós que representam os processos e arcos que representam o fluxo de dados.

O modelo funcional se preocupa com os valores de entrada e saída das funções, com o que acontece dentro da função, é usado para descrição dos processamentos e cálculos realizados, sendo, particularmente útil no desenvolvimento de cálculos de engenharia.

Processos: Transformam os dados, são implementados pelos métodos.

Fluxos: movem os dados.

Atores (objetos): Produzem e consomem dados.

Depósitos: Armazenam dados.

Exemplo:

Capítulo 56

Etapas de Desenvolvimento de Um Programa

Apresenta-se a seguir uma lista de etapas a serem seguidas no desenvolvimento de qualquer programa. Observe que sempre que se desenvolve um programa, estas etapas estão presentes, mesmo que não sejam documentadas.

Especificação do software. Descrição do objetivo e do que se espera do programa.

Análise Orientada a objeto do problema com o objetivo de identificar os objetos, os atributos e os métodos e a montagem da estrutura de relacionamento das classes. Pode ser utilizada uma ferramenta CASE. Uma descrição da modelagem TMO é descrita no capítulo 55.

Projeto do sistema. Definição e decisão dos conceitos relativos ao sistema a ser implementado. Escolha e definição da plataforma de programação: hardware, sistema operacional, linguagem e bibliotecas.

Projeto Orientado a objeto. Acréscimo a análise desenvolvida das características plataforma escolhida, maior detalhamento do funcionamento do programa.

Implementação do programa. Transformação do projeto em código. Integrar os diversos módulos, compilar, linkar.

Teste e depuração. Testar o programa realizando as tarefas usuais e depois as excepcionais. A medida que testa o programa corrige os erros encontrados.

Manutenção do programa. Incluir aperfeiçoamentos, corrigir problemas.

Documentação. Das especificações, dos assuntos, das classes, das relações, dos métodos e dos atributos. Criação do arquivo de help e dos manuais do programa.

56.1 Especificação

O desenvolvimento de um software inicia com a definição das especificações. As especificações ou o enunciado do problema deve ser gerado pelos clientes conjuntamente com os desenvolvedores.

As especificações definem as características gerais do programa, aquilo que ele deve realizar, e não a forma como irá fazê-lo. *Define as necessidades a serem satisfeitas.*

A primeira tarefa é definir *os objetivos do programa*. O contexto da aplicação, os pressupostos a serem respeitados e as necessidades de desempenho.

Depois deve-se iniciar a especificações do que se deseja do programa. Envolve a seleção do *tipo de interface*, a forma de *interação com o usuário*. Se a interface será de caracteres ou usando um ambiente gráfico. Se o programa poderá imprimir seus resultados (numa impressora), se salvará os resultados em disco, o formato de arquivo de disco. Se vai existir um HELP, e seu formato. Se vai ser de uma ou múltiplas janelas. Podem ser especificadas *características de desempenho*. O cliente define o que deve obrigatoriamente ser satisfeito e o que é opcional, enfim tudo que o software deve ser.

As especificações devem ser bem feitas, porque são a base para a etapa de análise orientada a objeto.

A seleção da plataforma de programação, que envolve a seleção do sistema operacional, da técnica de programação a ser adotada (aqui orientada a objeto) e da linguagem de programação só deve ser adotada após a etapa da análise abaixo descrita.

56.2 Análise orientada a objeto (AOO)

A segunda etapa do desenvolvimento de um POO, é a *Análise Orientada a Objeto (AOO)*. A AOO usa alguns conceitos chave anteriormente descritos e algumas regras para identificar os objetos de interesse. As relações entre as classes, os atributos, os métodos, as heranças e as associações.

A análise deve partir das especificações do software e de bibliotecas de classes existentes.

O modelo de análise deve ser conciso, simplificado e deve mostrar o que deve ser feito, não se preocupando como.

Segundo a técnica de Booch (1989), a análise pode ser iniciada a partir das especificações do programa: Os substantivos são possíveis classes e os verbos possíveis métodos.

O resultado da análise é um diagrama que identifica os objetos e seus relacionamentos.

A análise pode ser desenvolvida usando-se a metodologia TMO (veja Capítulo 55). Neste caso vamos ter três modelos, o modelo de objetos, o modelo dinâmico e o modelo funcional. Apresenta-se a seguir o que você deve realizar em cada um destes modelos.

56.3 Modelagem de objetos

Apresenta-se a seguir um conjunto relativamente grande de regras que existem para auxiliar o programador na identificação dos diversos componentes do programa, a identificação de assuntos, classes, objetos, associações, atributos operações e heranças. Estas regras existem para auxiliar você e esclarecer suas dúvidas.

A medida que o programador adquire experiência, ele não precisa ficar conferindo estas regras.

Sinais de associações desnecessárias: Não existem novas informações, faltam operações que percorram uma associação. Nomes de papéis abrangentes demais ou de menos. Se existe a necessidade de se obter acesso a um objeto por um de seus valores de atributos, considere a associação qualificada.

56.3.1 Identificação de assuntos

Um assunto é aquilo que é tratado ou abordado numa discussão, num estudo. Um assunto é usado para orientar o leitor em um modelo amplo e complexo.

Uma boa maneira de identificar os assuntos é dar uma olhada nos livros da área.

Preste atenção em semelhanças na forma de cálculo, procedimentos semelhantes indicam polimorfismo e são candidatas a superclasses. Usualmente, a classe mais genérica de um conjunto de classes identifica um assunto (a superclasse).

Exemplo:

- Num programa de análise de imagens de meios porosos, os assuntos podem representar:
Uma imagem obtida com o microscópio eletrônico de varredura; uma imagem binarizada;
Uma imagem caracterizada; uma imagem reconstruída; uma imagem simulada.

56.3.2 Identificação de classes

Segundo Boock (1989), para encontrar candidatas a classes, pegue as especificações e sublinhe os substantivos. Depois faça uma análise mais detalhada das possíveis classes, eliminando as desnecessárias e acrescentando alguma outra que tenha surgido. Segundo o autor, os verbos nas especificações costumam referenciar operações.

Johnson e Foote (1989) examinaram a questão de quando criar uma classe:

- A nova classe representar uma abstração significativa para o domínio do problema.
- Modelar com classes as entidades que ocorrem naturalmente no domínio do problema.
- Os métodos da classe forem provavelmente usados por várias outras classes.
- O seu comportamento for inerentemente complexo.
- A classe ou método fizer pouco uso das representações dos seus operandos.
- Se representada como um método de uma outra classe, poucos usuários desta classe a solicitariam.

Conservação das classes corretas²:

- **Classes redundantes:** Se duas classes expressarem a mesma informação o nome mais descritivo deve ser mantido.
- **Classes irrelevantes:** Se uma classe tiver pouco ou nada a ver com o problema deve ser eliminada.
- **Classes vagas:** Se a classe for muito vaga, deve ser encaixada em outra.
- **Atributos:** Nomes que descrevem principalmente objetos isolados devem ser considerados atributos. Se a existência independente de uma propriedade for importante transforme numa classe.

- **Operações:** Se um nome descreve uma operação que é aplicada a objetos e não é manipulada em si mesma, então não é uma classe.
- **Papéis:** O nome de uma classe deve refletir sua natureza intrínseca e não o papel que ela desempenha em uma associação.
- **Construções de implementação:** As construções inadequadas ao mundo real devem ser eliminadas do modelo de análise. Poderão ser usadas no projeto mas na não análise .

56.3.3 Identificação de objetos

Um objeto é simplesmente alguma coisa que faz sentido no contexto de uma aplicação.

Um objeto é um conceito, uma abstração, algo com limites nítidos e significado em relação ao problema.

56.3.4 Identificação de associações

Qualquer dependência entre duas ou mais classes é uma associação. Uma referência de uma classe a outra é uma associação.

As associações correspondem muitas vezes a verbos estáticos ou locuções verbais. Isso inclui a localização física (*junto á, parte de, contido em*), ações diretas (*direciona*), comunicação (*fala a*), propriedade (*tem, parte de*), ou satisfação de alguma condição (*trabalha para, casado com, gerencia*).

As associações podem ser implementadas de várias maneiras, mas as decisões de implementação devem ser mantidas fora do modelo de análise.

Depois de definidas as associações, deve-se verificar quais foram definidas incorretamente e descartá-las segundo os critérios abaixo:

Conservação das associações corretas²:

- **Ações:** Uma associação deve descrever uma propriedade estrutural e não um evento transiente.
- **Associações ternárias:** As associações entre três ou mais classes podem em sua maioria, ser decompostas em associações binárias ou expressas como associações qualificadas.
- **Associações derivadas:** Omita as associações que possam ser definidas em termos de outras associações porque seriam redundantes. Tanto quanto possível, classes, atributos e associações do modelo de objetos devem representar informações independentes.
- **Associações com nomes inadequados:** Não diga como nem porque uma associação ocorreu, diga o que ela é.
- **Nomes de papéis:** Ponha nomes de papéis onde forem adequados.
- **Associações qualificadas:** Um qualificador identifica os objetos do lado "muitos" de uma associação.

56.3.5 Identificação de atributos

Os atributos devem ter nomes significativos, devem representar uma propriedade do objeto, do domínio da aplicação ou do mundo real.

Os atributos geralmente correspondem a substantivos seguidos por frases possessivas, como "a cor do carro" ou "a posição do cursor". Os adjetivos muitas vezes representam valores de atributos específicos e enumerados, como vermelho, sobre ou expirado.

Alguns atributos podem ser derivados (quando são obtidos de outros atributos), a idade pode ser obtida da data atual do sistema e da data de nascimento da pessoa. Os atributos derivados devem ser diferenciados dos demais (por alguma notação), mas não devem ser omitidos. Eventualmente, na etapa de projeto poderão ser especificados através de um método.

Os atributos de ligação devem ser claramente identificados, pois existem em função de uma ligação entre 2 ou mais objetos. Os atributos de ligação devem ser claramente identificados.

Conservação dos atributos corretos²

- **Objetos:** se a existência independente de uma entidade for importante e não apenas o seu valor, então ela é um objeto.
- **Identificadores:** As linguagens baseadas em objetos incorporam a idéia de um identificador de objetos para fazer referência a um objeto sem ambigüidades. Não indique os identificadores no modelo de objetos.
- **Atributos de ligação:** Se uma propriedade depende da presença de uma ligação, então a propriedade é um atributo da ligação e não um objeto relacionado.
- **Valores internos:** Se um atributo descreve o estado interno de um objeto que é invisível fora do objeto, então elimine-o da análise. Ex: flags internos.
- **Refina os detalhes:** Omita os atributos menores que tem pouca probabilidade de afetar a maioria das aplicações.
- **Atributos discordantes:** Se alguns atributos parecem discordar dos demais, isto pode indicar a necessidade de se subdividir a classe em duas.

56.3.6 Identificação de heranças

Neste ponto, pode-se refinar o modelo incluindo os conceitos de herança. Realize primeiro a generalização e depois a especialização.

A generalização (top-down), pode ser verificada através de frases substantivas compostas por diversos adjetivos relativos ao nome da classe (Exemplo: Lâmpada incandescente, lâmpada fluorescente).

A especialização (botton-up) é realizada identificando-se atributos e operações semelhantes.

Na especificação podem ter sido definidos alguns sub-casos candidatos a herança.

Caminhos para elaborar classes abstratas²

- **Identificar mensagens e métodos comuns e migrá-los para uma superclasse.** Isto pode criar a necessidade de quebrar métodos e dividi-los entre superclasses e subclasses.

- Eliminar os métodos de uma superclasse que são freqüentemente sobrescritos em vez de herdados por suas superclasses. Isto torna a superclasse mais abstrata e conseqüentemente mais útil.
- Acessar todas as variáveis somente pelo envio de mensagens. As classes ficarão mais abstratas quando dependerem menos das suas representações de dados.
- Trabalhar subclasses para serem especializadas. uma subclasse será especializada se herdar todos os métodos da superclasse e acrescentar novos a si própria. Uma subclasse sempre representa um superconjunto da superclasse.
- O conceito de fatoração envolve a criação de sub-rotinas que serão acessadas por um método da classe base e serão implementadas de forma diferente por classes herdeiras.
- Subdivida uma função em sub-rotinas que serão diferentes para as diferentes classes herdeiras.
- Encapsule códigos externos. Se você deseja reaproveitar por exemplo uma biblioteca de matrizes desenvolvida em C, crie um objeto matriz que acesse as funções desenvolvidas em C.

56.3.7 Identificação de métodos (operações)

A inclusão de operações pode ser realizada a todo instante e geralmente é realizada baseada em um dos conceitos abaixo expostos.

Operações provenientes do modelo de objetos: as operações provenientes da estrutura de objetos incluem a leitura e a impressão de valores de atributos e as ligações em associações.

Operações provenientes de funções: Cada função no diagrama de fluxo de dados corresponde a uma operação em um objeto.

Simplificação das operações: Examine o modelo de objetos em busca de operações semelhantes e variações na forma de uma única operação, nestes casos procure utilizar a herança.

Operações provenientes de eventos²: Cada evento enviado a um objeto corresponde a uma operação no objeto. Se for feito um modelo dinâmico, os eventos não precisam ser listadas no modelo de objetos.

Operações provenientes de ações e de atividades de estados²: As ações e atividade do diagrama de estados podem ser funções.

Conservação dos métodos corretos²:

- Projetar os métodos com um único objetivo.
- Projetar um novo método quando se defrontar com a alternativa de ampliar um já existente.
- Evitar métodos extensos (máximo 30 linhas).
- Armazenar como variáveis de classe as variáveis que são necessárias a mais de um método ou a uma subclasse.

56.3.8 Teste dos caminhos de acesso

Teste os diversos caminhos do modelo obtido, para verificar sua consistência e completeza.

56.3.9 Iteração

Esta etapa consiste em repetir as diversas etapas anteriormente realizadas, com o objetivo de encontrar e eliminar erros, de lembrar pontos esquecidos e de verificar a coerência do modelo.

Procure por erros como:

- Assimetria nas associações e generalizações
- Atributos e operações incompatíveis
- Dificuldade em realizar uma generalização
- Uma operação sem boas classes alvo
- Associações repetidas
- Classes desnecessárias (sem atributos, métodos)
- Métodos sem caminhos de acesso indicam a falta de associações

56.3.10 Preparação do dicionário de dados

Identificadas as classes e objetos, você deve criar um dicionário de dados, com o nome de cada classe/objeto e a descrição em um parágrafo do que é e representa.

56.4 Modelagem dinâmica²

Lembre-se, este é um título de nível 2, só deve ser lido por usuários intermediários ou avançados.

Apresenta-se a seguir um conjunto de dicas para implementação do modelo dinâmico.

56.4.1 Formação de interfaces

Para formar a interface de seu programa, parta da interface de programas existentes, das especificações do programa, do conhecimento dos usuários e procure fazer uma interface o mais simples possível. Deve-se testar a interface.

A interface é geradora de muitos eventos e deve servir de base para a montagem dos diversos cenários.

56.4.2 Preparação de um cenário

Um cenário deve representar uma seqüência típica de uso do programa, ou seja, a execução de determinadas tarefas padrões. Também devem representar as exceções, casos em que o usuário comete algum erro, casos em que o sistema não consegue realizar as tarefas solicitadas. Devem ser montados diversos cenários.

56.4.3 Identificação de eventos

Os eventos incluem toda e qualquer interação do usuário com o programa (seleções de menu, entrada de dados, pressionamento do mouse,...). Decisões, interrupções, transições, ações de ou para usuários de dispositivos externos.

Agrupe sob um único nome os eventos que tem o mesmo efeito sobre o fluxo de controle, mesmo se os valores dos parâmetros diferem.

Prepare um diagrama de eventos para cada cenário

Para cada cenário, crie diagramas de eventos iniciais. Listando os objetos e os diversos eventos que partem de um objeto para outro.

56.4.4 Construa um diagrama de estados

Prepare um diagrama de estados para cada classe de objetos com comportamento dinâmico importante (não trivial), mostrando os eventos que o objeto recebe e envia.

Inicie a construção do diagrama de estados a partir do diagrama de eventos oriundos dos cenários.

Todo cenário corresponde a um caminho a ser seguido no diagrama de estados, ou seja deve-se comparar os diversos cenários e verificar os pontos onde eles divergem (e que precisam ser codificados no diagrama de estados). Lembre-se que dois caminhos num diagrama de estados serão os mesmos se o objeto esquecer os valores passados.

Depois de considerar os eventos normais (default) considere as exceções (casos de erro).

De a cada estado um nome significativo. O nome pode não ser necessário se for diretamente identificado.

Se a seqüência puder ser repetida indefinidamente ela forma um loop. Sempre que possível substitua seqüências finitas por loop's.

Para ajudar na identificação dos estados, faça uma análise de cada atributo do objeto. Ou seja verifique se todos os atributos estão representados no diagrama de estados.

Lembre-se que uma instrução **faça: X** em um objeto pode ser um evento para outro objeto, ou seja verifique as instruções **faça:** nos diversos objetos e verifique se a mesma não representam um evento para outro objeto, se representar desenhe no outro objeto.

56.4.5 Compare eventos entre objetos para verificar a consistência

Depois de montado o diagrama de estados é necessário verificar a consistência do mesmo. Verificar se esta completo e se é consistente.

Verifique os erros de sincronização, quando uma entrada ocorre em momento inadequado.

56.5 Modelagem funcional³

O modelo funcional mostra como os valores são processados, sem considerar a seqüência, as decisões ou a estrutura de objetos. O modelo funcional mostra quais valores dependem de outros e as funções que os relacionam.

As funções são expressas de várias maneiras, incluindo a linguagem natural, equações matemáticas e pseudo código.

Identificação de valores de entrada e saída: Comece identificando os parâmetros de entrada e de saída.

Construção do diagrama de fluxo de dados: Construa um diagrama de fluxo de dados, mostrando como cada atributo é manipulado e alterado. Se um determinado processo for complexo, ele pode ser subdividido num segundo nível. Os diagrama de fluxo de dados devem especificar somente dependências entre operações.

Descrição das funções: Somente depois de refinar o diagrama de fluxo de dados, você deve descrever cada função. Concentre o foco no que a função faz, não em como implementá-la.

Identificação de restrições entre objetos: Identifique as restrições ente objetos. As restrições são dependências funcionais entre objetos que não são relacionadas por uma dependência de entrada/saída. As pré-condições em funções são restrições que os valores de entrada devem satisfazer, e as pós-condições são restrições que os valores de saída devem conservar.

Epecificação de critérios e otimização: Deve-se determinar quais os pontos em que deve haver otimização.

56.6 Projeto do sistema

Depois da análise orientada a objeto desenvolve-se o projeto do sistema. Nesta etapa são tomadas decisões de alto nível relativas ao sistema a ser implementado.

Deve-se definir padrões de documentação, nome das classes, padrões de passagens de parâmetros em funções, padrões de retorno de funções, características da interface do usuário, características de desempenho.

O projeto do sistema é a estratégia de alto nível para resolver o problema e elaborar uma solução, envolve etapas como a subdivisão do sistema em subsistemas, a alocação dos subsistemas ao hardware e software, a tomada de decisões conceituais e políticas que formam a infra-estrutura do projeto detalhado.

56.6.1 Interface interativa²

Uma interface interativa é dominada pelas interações entre ela e agentes externos, como pessoas, dispositivos e outros programas. O principal aspecto é o protocolo de comunicação entre o sistema e os agentes externos. As interfaces interativas são dominadas pelo modelo dinâmico.

Etapas: Isole os métodos que formam a interface dos objetos e que definem a semântica da aplicação. Utilize objetos predefinidos para interagirem com agentes externos. Utilize o modelo dinâmico como estrutura do programa. Separe os eventos físicos dos eventos lógicos. Especifique de forma completa as funções aplicativos que são convocadas pela interface.

56.6.2 Simulação dinâmica

Usada para modelar ou delinear objetos do mundo real. Talvez sejam os sistemas mais simples de projetar usando a modelagem orientada a objeto.

Etapas: Identifique os atores, objetos ativos do mundo real e do modelo de objetos. Identifique os eventos discretos. Identifique dependências contínuas. Geralmente uma simulação é controlada por um loop de tempo em uma fina escala de tempo.

56.6.3 Identificação de subsistemas²

Um subsistema é um conjunto de classes, associações, operações, eventos e restrições inter-relacionadas, que tem uma interface razoavelmente bem definida.

Cada subdivisão do sistema deve englobar, propriedades comuns (funcionalidade similar, a mesma localização física, algumas propriedades comuns).

A interface especifica a forma das interações e o fluxo das informações através das fronteiras, mas não especifica como o programa é especificado internamente.

Os subsistemas de mais baixo nível são denominados de módulos.

O relacionamento entre os subsistemas pode ser do tipo cliente-servidor ou homogêneo.

Camadas: Um subsistema conhece as camadas que estão abaixo dele mas desconhece as camadas superiores. As arquiteturas em camadas existem em duas formas: fechada e aberta. Fechada quando é construída apenas em termos das camadas imediatamente abaixo.

56.6.4 Identificação de concorrências²

O modelo dinâmico é o guia para a identificação de concorrências.

Identificação de concorrências inerentes: Dois objetos são inerentemente concorrentes se puderem receber eventos ao mesmo tempo sem interagirem.

Definição de tarefas concorrentes: Embora todos os objetos sejam conceitualmente concorrentes, na prática muitos objetos de um sistema são interdependentes.

56.6.5 Uso dos processadores²

Estimativa da necessidade de recursos de hardware. Alocação de tarefas a processadores. Determinação da conectividade física.

56.6.6 Identificação de depósitos de dados²

Vantagens do uso de bancos de dados: Quando existe compartilhamento por muitos usuários ou aplicações, a distribuição de dados, a melhoria da integridade, da extensibilidade e do suporte de transações pode ser obtida com o uso de banco de dados. Os bancos de dados apresentam interface comum para diferentes aplicações, uma linguagem de acesso padronizada.

Desvantagens do uso de bancos de dados: Sobrecarga de desempenho, funcionalidade insuficiente para aplicações avançadas, interface desajeitada com as linguagens de programação. Necessidade de controle da base de dados.

O gerenciador de transações: Um gerenciador de transações é um sistema de banco de dados cuja principal função é armazenar e acessar informações.

56.6.7 Manipulação de recursos globais²

O projetista deve identificar recursos globais e formular mecanismos de acesso e compartilhamento destes recursos. Cada objeto global deve ser propriedade de um objeto guardião que controla o acesso a ele (Exemplo: Uma impressora de rede.)

56.6.8 Escolha da implementação de controle²

O controle pode ser externo e interno. O controle externo é o fluxo de mensagens externamente visível entre os subsistemas. Existem três tipos de controle:

Sistema baseado em procedimentos: O controle reside no código do programa. É adequado se o modelo de estados mostrar uma alternância regular de eventos de entrada e saída.

Sistema baseado em eventos: Os eventos são diretamente manipulados pelo despachante, é mais simples e poderoso. Exemplo: Programas para windows.

Sistemas concorrentes: O controle reside de modo concorrente em diversos objetos independentes, sendo cada um uma tarefa separada.

56.6.9 Manipulação de condições extremas²

As condições extremas envolvem a inicialização a destruição e as falhas em objetos.

56.6.10 Estabelecimento de prioridades

Exemplo: o importante é velocidade, depois memória, depois portabilidade e custo.

56.6.11 Estruturas arquitetônicas comuns³

Lista-se abaixo as estruturas arquitetônicas comuns.

Transformação em lote: Uma transformação em lote é uma transformação seqüencial entrada/saída.

Transformação contínua: Uma transformação contínua é um sistema em que as saídas dependem ativamente da modificação das entradas e devem ser periodicamente atualizadas.

Sistemas em tempo real: É um sistema interativo em que as restrições de tempo nas ações são muito rígidas, não pode ser tolerada a menor falha de tempo.

56.7 Projeto orientado a objeto²

O projeto orientado a objeto, é a etapa que segue o projeto do sistema. Se baseia na análise, mas leva em conta as decisões do projeto do sistema. *Objetiva acrescentar a análise desenvolvida, as características da linguagem de programação e da plataforma escolhida, ou seja detalhes de implementação.* Passa pelo maior detalhamento do funcionamento do programa, acrescentando atributos e métodos que envolvem a solução de problemas específicos, não definidos durante a análise.

Envolve a otimização da estrutura de dados e dos algoritmos, minimização do tempo de execução, memória e custos. Existe um desvio de ênfase para os conceitos de computador. Pode-se acrescentar ainda rotinas com o objetivo de melhorar o desempenho do soft.

Por exemplo: Na análise você define que existe um método para salvar um arquivo em disco, define um atributo nomeDoArquivo, mas não se preocupa com detalhes específicos da linguagem. Já no projeto você inclui as bibliotecas necessárias para acesso ao disco, cria um atributo específico para acessar o disco. Podendo portanto acrescentar novas classes aquelas desenvolvidas na análise.

Com as ferramentas CASE existentes como o programa With Class, pode-se gerar imediatamente o código do programa a partir do projeto. Obviamente é um código inicial, que exige a interação do programador, na definição efetiva dos diversos métodos.

56.7.1 Implementação do controle

- Controle no interior de um programa.
- Controle com máquinas de estado.
- Controle com tarefas concorrentes.

56.7.2 Métodos->localização

Definir em que classe uma determinada operação deve ser colocada nem sempre é um processo fácil. Se o objeto é um objeto real, é fácil identificar suas operações, mas existem algumas operações internas de difícil localização. A pergunta é onde devo colocar esta função?. Em geral uma operação deve ser colocada na classe alvo. Se existe uma classe mais fortemente afetada por determinada operação, esta deve ser colocada nesta classe.

56.7.3 Métodos->otimização de desempenho

Lista-se abaixo algumas tarefas que podem ser realizadas com o objetivo de melhorar o desempenho de um soft.

- O acréscimo de associações redundantes para eficiência de acesso
- A reorganização da ordem de execução para melhoria de desempenho
- A salvaguarda de atributos derivados para evitar o reprocessamento

56.7.4 Ajustes nas heranças

Reorganização das classes e das operações (criar funções genéricas com parâmetros que nem sempre são necessários e englobam funções existentes).

Abstração do comportamento comum (duas classes podem ter na realidade uma superclasse em comum).

Utilização de delegação para compartilhar a implementação (quando você cria uma herança irreal, para reaproveitar código. Não recomendável).

56.7.5 Ajustes nas associações

Deve-se definir na fase de projeto como as associações serão implementadas, se obedecerão um determinado padrão ou não.

Associações unidirecionais: Se a associação é unidirecional, pode ser projetada como um ponteiro em uma das classes. Se for uma associação "um" pode ser implementada como um

ponteiro; se for "muitos" pode ser implementada como um conjunto de ponteiros; se for "muitos ordenada" pode ser implementada como um único ponteiro (lista).

Associações bidirecionais: Se a classe A acessa mais vezes a classe B, pode ser um ponteiro em A. Quando B precisar acessar A pode-se realizar uma pesquisa.

Se os acessos em A e B foram em mesmo número, deve-se implementar ponteiros em ambos os objetos.

Se existe uma relação de "muitos", pode-se implementar com a utilização de um dicionário, que é uma listagem de objetos associação que relaciona os dois objetos. Assim o objeto A acessa o objeto dicionário e este tem o endereço correto do objeto B, e o objeto B acessa o objeto dicionário que tem o endereço correto do objeto A correto.

56.7.6 Ajustes nos atributos de ligação

São os atributos que só existem em função da ligação entre duas classes. Para identificar onde colocar o atributo obedeça o seguinte.

Se a associação for um-para-um o atributo pode ser localizado em qualquer dos objetos.

Se a associação for um-para-muitos, o atributo deve ser localizado no lado muitos.

Se a associação for muitos-para-muitos, deve-se criar uma classe para a associação.

56.7.7 Empacotamento físico

Ocultamento de informações internas da visão externa (caixa preta), minimizando as dependências entre os diversos módulos.

Evite a chamada de um método em função do resultado de outro método.

Evite percorrer associações para acessar dados de classes distantes.

Coerência de entidades. Um objeto é coerente se estiver organizado com um plano consistente e se suas partes tiverem um objetivo comum. Se uma classe for muito complexa deve ser dividida.

Construção de módulos. Se um grupo de classes trocam muitas informações entre si, elas fazem parte de um módulo do programa. Provavelmente um assunto.

56.7.8 O projeto de algoritmos

No modelo funcional são especificadas todas as operações e o que devem fazer, o algoritmo mostra como fazer.

A escolha de algoritmos: Deve-se implementar os algoritmos da forma mais simples possível, conservando a facilidade de implementação e de compreensão. O algoritmo deve ser flexível, o que significa que possa ser alterado posteriormente, sem grandes dificuldades. Em alguns casos deve-se criar um nível a mais com o objetivo de deixar o algoritmo mais genérico e útil.

A escolha de estruturas de dados: As estruturas podem ser arrays, listas, pilhas, filas, conjuntos, dicionários, associações, árvores, e outras variações.

56.8 Implementação

Com o código inicial do programa gerado por uma ferramenta CASE como o With Class ou o AppExpert do Borland C++, parte-se para a implementação do programa. Nesta etapa são essenciais não só os conhecimentos da filosofia orientada a objeto, mas da linguagem de programação.

Ou seja, as regras de sintaxe e a forma como a linguagem implementa a programação orientada a objeto.

A medida que se implementa o código, as diversas classes e métodos; pode-se testar cada módulo desenvolvido.

56.9 Testes

O teste se preocupa com o funcionamento lógico do programa, durante o teste do programa você deve verificar conceitos lógicos.

Primeiro testar os casos simples, mais usuais. Depois os casos complexos, com as variáveis assumindo valores perto dos extremos admitidos. Deve-se testar cada classe criada, para tal pode-se criar pequenos programas de teste.

56.10 Documentação de um programa

A documentação de um programa é essencial pelos seguintes motivos:

- Compreensão do funcionamento do programa e de seu **planejamento**.
- Acompanhamento da **execução** das atividades de implementação, testes e depuração.
- Compreensão e **controle** das atividades desenvolvidas.
- Preparação dos **manuais** do programa.
- Preparação do **help** do programa.
- Permitir a **manutenção** e alteração do programa por terceiros.

Deve-se criar um diretório, onde serão armazenados os arquivos do programa a ser desenvolvido. Dentro deste diretório crie um diretório DOCUMENTAÇÃO, onde serão incluídas todas as informações relativas ao programa, ou seja, a documentação do sistema, das classes, e das bibliotecas desenvolvidas. O arquivo de help do programa. Documentação das decisões de projeto. Arquivo com os bugs (identificados/solucionados), arquivo LEIAME, change.log, INSTALL.

A documentação é desenvolvida ao longo do desenvolvimento do programa e deve servir de base para o desenvolvimento dos manuais, estes devem ser desenvolvidos somente após a conclusão do programa.

Abaixo apresenta-se os diferentes tipos de documentação que devem ser desenvolvidos, observe que eles seguem uma hierarquia.

56.10.1 Documentação do sistema

Nome do sistema:

Sistemas inclusos ou subsistemas:

Responsabilidades:

Formas de acesso:

Bibliotecas utilizadas:

Diversos:

56.10.2 Documentação dos assuntos

Nome do assunto ou área:

Descrição do que representa:

Acessos:

Bibliotecas:

56.10.3 Documentação das classes

Nome da classe:

Descrição do objetivo:

Assunto a que esta relacionada:

Superclasse:

Acesso:

[Cardinalidade das relações:]

[Concorrência:]

[Transformações:]

[Especificações da linguagem:]

[Persistência (se é armazenado em disco):]

[Tamanho:]

[Abstrata/Concreta:]

[Arquivo de documentação auxiliar:]

56.10.4 Documentação das relações

Descrição:

Cardinalidade:

Atributo atravessado:

Tipo de relação:

Diversos:

56.10.5 Documentação dos atributos

Nome: Descrição:

Tipo: Valor inicial: Valor mínimo: Valor máximo:

Restrições:

Derivado(Y/N):

Linguagem (friend/ const /static):

Tamanho:

56.10.6 Documentação dos métodos

Nome: Descrição:

Retorno:

Parâmetros:

Acesso:

Pré-condições:

Abstrato/Normal:
Exceções:
Concorrência:
Tempo de processamento:
Tamanho:

56.11 Manutenção

A manutenção envolve o conceito de manter o programa atualizado. A medida que o tempo passa, novas exigências (especificações) são realizadas pelos usuários e o programador deve modificar o programa com o objetivo de dar resposta as novas necessidades do usuário.

56.11.1 Extensibilidade, robustes, reusabilidade²

Extensibilidade²: Um programa sempre é aperfeiçoado por você posteriormente ou por terceiros. Para facilitar a extensão de um programa procure:

- Encapsular classes.
- Ocultar estruturas de dados.
- Evitar percorrer muitas associações ou ligações.
- Evitar instruções "case" sobre o tipo de objeto.
- Distinguir funções públicas e privadas.

Como contruir métodos robustos²: Um método é robusto se ele não falha, mesmo quando recebe parâmetros errados. Os métodos que são acessados diretamente pelo usuário (interface) devem ter um controle dos parâmetros fornecidos pelo usuário. Os métodos internos não necessitam de verificação de parâmetros (pois consome mais tempo de processamento), entretanto, se existirem algum métodos críticos estes podem ter algum tipo de verificação. Para que o programa seja mais robusto:

- Só otimize o programa depois de o mesmo funcionar e ter sido testado.
- Valide argumentos de métodos acessados pelo usuário.
- Não inclua atributos que não podem ser validados.
- Evite limites pré-definidos. Dê preferência a alocação dinâmica de memória.
- **Reduzir o número de argumentos, dividindo uma mensagem em várias (Número argumentos ≤ 6).**
- **Reduzir o tamanho dos métodos para até 30 linhas.**
- Durante a fase de desenvolvimento inclua instruções que facilitem a operação e acompanhamento do desenvolvimento do programa.

Reusabilidade²: Consiste em montar um programa com a preocupação do mesmo ser posteriormente reaproveitado. Exige a necessidade de se deixar o programa mais genérico. Algumas regras para aumentar a reusabilidade:

- Manter os métodos pequenos (com menos de 30 linhas de código).
- Manter os métodos coerentes (executa um única função ou funções estreitamente relacionadas).
- Manter os métodos consistentes (métodos semelhantes devem ter nomes semelhantes, e formatos semelhantes).
- Separar métodos políticos (aqueles que envolvem a tomada de decisões) dos de implementação (aqueles que realizam um procedimento específico).
- Os parâmetros dos métodos devem ser passados de forma uniforme.
- Deixe o método o mais genérico possível.
- Não acesse informações globais em um método.
- Evite métodos que mudam seu comportamento drasticamente em função de alterações do contexto do programa.

Aperfeiçoamento da qualidade dos protocolos padrões²: Dar nomes similares ou idênticos a mensagens e métodos quando uma classe comunica-se com outras classes para realizar operações similares.

Trabalhar qualquer código que claramente cheque a classe de um objeto e desenhar classes onde uma mensagem possa ser enviada diretamente para um objeto e manipulada corretamente pelos métodos nele contidos.

Agrupamento de classes em módulos (ou assuntos), gerando classes abstratas²: Depois de corrigido o modelo, pode-se agrupar as classes em folhas e módulos que tem algum sentido lógico. Normalmente um associação só aparece em uma única folha, já uma classe pode aparecer em mais de uma folha, mostrando a ligação entre as diferentes folhas.

Identificação de bibliotecas²: É muito importante que um objeto seja completo, para que possa ser utilizado como uma biblioteca expansível. Só necessitando ser recompilado. Nenhuma variável interna deve ser acessada pela função mãe pelo nome, a função mãe deve passar as informações para dentro do objeto onde são tratadas.

Identificação de Framework²: Uma biblioteca de classes é algo genérico, com a classe base (superclasse), as classes filhas, os atributos e métodos básicos. Além da estrutura de ligação das classes.

Já uma Framework é uma biblioteca de classes que foi aprimorada, aperfeiçoada para solucionar os problemas específicos de uma determinada área.

Uma biblioteca de classes precisa ser desenvolvida, planejada, a mesma não surge espontaneamente.

As frameworks são o objetivo fundamental do projeto orientado a objeto, por representarem o nível mais alto de abstração.

Identificar subclasses que implementem o mesmo método de diferentes maneiras. Se um método é sempre redefinido, reconsidere onde estes métodos poderiam estar mais bem localizados.

Identificar e dividir classes em que alguns métodos acessam somente algumas variáveis de instância e outros métodos acessam somente as outras variáveis de instância.

Enviar mensagens para outras classes em vez de para a própria classe. Substituir frameworks baseadas em hereditariedade por frameworks baseadas em componentes, sobrepondo métodos com mensagens enviadas para os componentes.

Identificar conjuntos de métodos combinados em uma classe somente para acessar uma variável de instância comum. Considerar a migração de um ou mais métodos para outras classe; mudar os métodos para passar parâmetros explícitos. Isto facilitará a divisão de classes.

Programação em grande escala³:

- Não inicie o programa prematuramente
- Mantenha os métodos compreensíveis
- Faça métodos legíveis
- Utilize os mesmos nomes do modelo de objetos
- Escolha os nomes cuidadosamente
- Utilize diretrizes (regras) de programação
- Procure criar módulos empacotando as classes
- Documente as especificações, classes, métodos atributos, e as associações
- Faça um documento de uso do programa pelo usuário

Empacotamento: O conceito de **empacotamento** envolve a necessidade de se unir dois ou mais programas desenvolvidos por pessoas diferentes. Pode ocorrer que os dois programadores desenvolveram classes com o mesmo nome e você terá a necessidade de alterar o nome de uma delas. Uma linguagem que permite um bom empacotamento elimina esta necessidade. Felizmente C++ fornece o conceito de namespace, facilitando um bom empacotamento.

Referências Bibliográficas

- [Ann L. Winblad, 1993] Ann L. Winblad, e. a. (1993). *Software Orientado a Objeto*, volume 1. Makron Books, São Paulo.
- [Bjarne, 1999] Bjarne, S. (1999). *C++ The Programming Language*, volume 1. John Wiley Sons, 3 edition.
- [Borland, 1996a] Borland (1996a). *Borland C++ Programers Guide*, volume 1. Borland, 5 edition.
- [Borland, 1996b] Borland (1996b). *Borland C++ Programers Guide*, volume 2. Borland, 5 edition.
- [Cederqvist, 1993] Cederqvist, P. (1993). *Version Management with CVS*. GNU.
- [Coad and Yourdon, 1993] Coad, P. and Yourdon, E. (1993). *Análise Orientada a Objeto*. Campus, São Paulo.
- [Cooper, 1999] Cooper, M. (1999). *Building and Installing Software Packages for Linux - HOWTO*.
- [Deitel and Deitel, 1997] Deitel, H. and Deitel, P. (1997). *C++ How to Program*. Prentice Hall, New Jersey, 2 edition.
- [Deitel and Deitel, 2001] Deitel, H. and Deitel, P. (2001). *Como programar em C++*. Bookman, Porto Alegre, 3 edition.
- [Dietz, 1998] Dietz, H. (1998). *Linux Parallel Processing HOWTO*. <http://yara.ecn.purdue.edu/pplinux/PPHOWTO/pphowto.html>.
- [Eckel, 2000] Eckel, B. (2000). *The STL made simple*. <http://www.MindView.net>.
- [Ezzel,] Ezzel, B. *Programação em Turbo C++: Uma Abordagem Orientada a Objeto*. Rio de Janeiro, 1991 edition.
- [Ezzel, 1991] Ezzel, B. (1991). *Programação Gráfica em Turbo C++: Uma Abordagem Orientada a Objeto*. Ciência Moderna, Rio de Janeiro.
- [Ezzel, 1993] Ezzel, B. (1993). *Programação em Window NT 3.1*. IBPI, Rio de Janeiro.
- [Gratti, 1999] Gratti, R. (1999). *Bibliotecas Compartilhadas*, volume 5. Revista do Linux.
- [Hughes and Hughes, 1997] Hughes, C. and Hughes, T. (1997). *Object Oriented Multithreading using C++: architectures and components*, volume 1. John Wiley Sons, 2 edition.

- [Jeff and Keith, 1993] Jeff, D. and Keith, W. (1993). *C/C++ Ferramentas Poderosas*. Berkeley, Rio de Janeiro.
- [Kurt Wall, 2001] Kurt Wall, . (2001). *Linux Programming Unleashed*, volume 1. SAMS, 2 edition.
- [Maguire, 1994] Maguire, S. (1994). *Guia Microsoft para o Desenvolvimento de Programas sem Erros*.
- [Manika, 1999] Manika, G. W. (1999). *Super-Computador a Preço de Banana*, volume 2. Revista do Linux.
- [Margaret and Bjarne, 1993] Margaret, E. and Bjarne, S. (1993). *C++ Manual de Referência Comentado*. Campus.
- [Martin and McClure, 1993] Martin, J. and McClure, C. (1993). *Técnicas Estruturadas e CASE*. MacGraw-Hill, São Paulo.
- [Nolden and Kdevelop-Team, 1998] Nolden, R. and Kdevelop-Team (1998). *The User Manual to KDevelop*. kdevelo team.
- [Pappas and Murray, 1993] Pappas, C. H. and Murray, W. H. (1993). *Turbo C++ Completo e Total*. Mac-Graw Hill, São Paulo.
- [Perry, 1995a] Perry, G. (1995a). *Programação orientada para objeto com turbo C++*. Berkeley.
- [Perry, 1995b] Perry, P. (1995b). *Guia de Desenvolvimento Multimídia*. Berkeley, São Paulo.
- [Radajewski and Eadline, 1998] Radajewski, J. and Eadline, D. (1998). *Beowulf HOWTO*. <http://www.sci.usq.edu.au/staff/jacek/beowulf/BDP>.
- [Raymond, 2000] Raymond, E. S. (2000). *Software Release Practice HOWTO*.
- [Roberto and Fernando, 1994] Roberto, P. and Fernando, M. (1994). *Orientação a objetos em C++*. Ciencia Moderna, Rio de Janeiro.
- [Rumbaugh et al., 1994] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1994). *Modelagem e Projetos Baseados em Objetos*. Edit. Campus, Rio de Janeiro.
- [Schildt, 1990] Schildt, H. (1990). *C Completo e Total*. Mcgraw-Hill, São Paulo.
- [Steven and Group, 1993] Steven, H. and Group, T. P. N. C. (1993). *Programando em C++*. Campus, Rio de Janeiro.
- [Swan, 1994] Swan, T. (1994). *Programação avançada em Borland C++ 4 para Windows*. Berkeley, São Paulo.
- [Vasudevan, 2001a] Vasudevan, A. (2001a). *C-C++ Beautifier HOWTO*.
- [Vasudevan, 2001b] Vasudevan, A. (2001b). *C++ Programming HOWTO*.
- [Wiener and Pinson, 1991] Wiener, R. S. and Pinson, L. J. (1991). *Programação Orientada para Objeto e C+*. MacGraw-Hill, São Paulo.

Parte VII

Apêndices

Apêndice A

Diretrizes de pré-processador

Neste capítulo apresenta-se as diretrizes de pré-processados.

A.1 Introdução as diretrizes de pré processador

As diretrizes de pré processador são orientações, instruções, que damos para a compilação do programa. Uma diretriz de pré-processador inicia com o símbolo `#` e é utilizada para compilação condicional, substituição de macros e inclusão de arquivos nomeados.

```
Exemplo:  
//Para realizar a inclusão de arquivos.  
#include <nome-do-arquivo>  
#include "nome-do-arquivo"
```

Use `<>` para incluir arquivos da biblioteca e `"` para incluir arquivos do diretório atual. A diferença é que a segunda instrução procura pelo arquivo nos diretórios do usuário e depois nos diretórios da path e do compilador.

A.2 Compilação condicional

A compilação condicional permite que sejam colocadas condições que restringem as regiões do arquivo a serem compiladas. A vantagem da compilação condicional é poder colocar instruções que diferenciam plataformas. Alguns arquivos podem ter nomes diferentes no Windows, no Linux/Unix ou no Mac OS X, usando a compilação condicional você pode verificar em que sistema esta e incluir os arquivos corretos.

Apresenta-se a seguir as instruções que podem ser utilizadas para compilação condicional.

A.2.1 if

Se a expressão for verdadeira o compilador verifica o código do bloco, se for falsa pula até o próximo `endif`

```
#if expressão_constante  
{bloco}  
#endif
```

A.2.2 if...else

Se a expressão `_constante1` for verdadeira o compilador verifica o `bloco1`, se for falsa executa o `bloco2` até o próximo `#endif`.

```
#if expressão_constante1
bloco1
#else
bloco2
#endif
```

Exemplo:

```
#include <iostream>
using namespace std;
#if 0
    void main(){cout<<"oi1\n";}
#else
    void main(){cout<<"oi2\n";}
#endif
```

A.2.3 if...elif...elif...endif

Para múltiplas verificações use:

```
#if expressão_constante1
bloco1
#elif expressão_constante2
bloco2
#elif expressão_constante3
bloco3
#endif
```

A.2.4 define, ifdef, ifndef, undef

A linguagem C permite a definição de variáveis a nível de compilador, variáveis que são passadas para o compilador para a realização do processo de compilação. As instruções `define` funcionam de um modo semelhante ao `search/replace` de seu editor de texto.

```
//define uma variável
#define variável
//trocar variável por valor
#define variável valor
//Se var já foi definida compile o bloco
#ifdef var
bloco
#endif
//Se var não foi definida compile o bloco
#ifndef var
```

```
bloco
#endif
//Apaga definição da variável var
#undef var
//Diretiva de erro
//Gera uma mensagem de erro
#error mensagem
```

A.2.5 Macros

Na linguagem C você pode usar macros, que são pequenas funções a nível de pré-processamento. Uma macro é implementada com uma diretiva define.

```
Exemplo:
//definição da macro
#define soma(x,y) x+y
//uso da macro
int a = 4; int b = 5;
int z = soma(a,b); //resultado int z = a + b;
```

Na linguagem C++, as macros foram substituídas pelas funções inline, que são mais eficientes por fazerem verificação de tipo.

Dica: Evite usar macros, as funções inline do C++ fazem a mesma coisa de forma mais inteligente.

Macros pré-definidas

`__LINE__` Número da linha compilada.
`__FILE__` Nome do arquivo sendo compilado.
`__DATE__` Data de compilação.
`__TIME__` Hora minuto e segundo.
`__STDC__` Se é compilador padrão.

Em nossos programas costumamos incluir algumas bibliotecas, como a `iostream.h`. Se tivermos um programa com muitos arquivos, a biblioteca `iostream.h` vai ser incluída em cada arquivo. Para evitar que a `iostream.h` seja incluída duas vezes na mesma unidade de tradução, existe na `iostream.h` as seguintes diretrizes de pré-processador.

```
Exemplo:
#ifndef __IOSTREAM_H
#define __IOSTREAM_H
//Seqüência da biblioteca....
#endif /* __IOSTREAM_H */
```

As instruções acima fazem o seguinte: Primeiro verifica se a variável `__IOSTREAM_H` já foi definida, se não definida então define a variável e compila a sequência da biblioteca. Se a mesma já foi definida, o bloco “//Sequência da biblioteca...” é pulado. Observe que com estas instruções o arquivo só é compilado uma única vez.

Dica: No BC5.0 a diretiva `#pragma hdrstop`, faz com que as bibliotecas antes de `hdrstop` façam parte do arquivo `*.csm`. que é um arquivo que guarda as informações da compilação para esta se processe mais rapidamente. É um arquivo muito grande.

Dica: Uma macro muito utilizada durante a depuração é a macro `assert`:

```
assert(ponteiro != NULL);
```

que encerra o programa se o ponteiro for nulo. `Assert` será discutida em detalhes no capítulo `Debug`.

Apêndice B

Conceitos Úteis Para Programação em C/C++

Neste capítulo apresenta-se as classes de armazenamento, os modificadores de acesso e o escopo das variáveis.

B.1 Classes de armazenamento²

A classe de armazenamento se refere ao tempo de vida do objeto.

A definição da classe de armazenamento vem antes da declaração da variável e instrue o compilador sobre o modo como a variável vai ser armazenada.

Existem 3 palavras chaves para definir a classe de armazenamento.

auto: É o método default para armazenamento de variáveis. Quando for encerrada a função ou bloco onde a variável foi declarada a mesma vai ser eliminada, isto é, deletada automaticamente.

register: Estamos pedindo para o compilador colocar a variável em um registro, que tem processamento mais rápido. Quando for encerrada a função ou bloco onde foi declarada, a variável vai ser eliminada. Os compiladores mais modernos colocam nos registradores as variáveis mais utilizadas.

static: A variável passa a existir a partir de sua definição e dura toda a execução do programa. Se não for definido um valor assume o valor 0 (zero). Os objetos estáticos são os primeiros a serem construídos e os últimos a serem destruídos em um programa. Todos as variáveis globais tem classe de armazenamento estática, isto é, existem por toda vida do programa. Se dentro de uma função houver uma variável static, a mesma será criada quando o programa passar pela função pela primeira vez, na segunda passagem a variável não é novamente criada. Um objeto ou membro de um objeto pode ser estático se for declarada explicitamente com static.

Exemplo:

```
int funcao()
{
//o mesmo que int x;
```

```

auto int x;
//colocar no registro
register int y;
//deixar na memória
static char Titulo[]="Pro-Image";
..
}

```

B.2 Modificadores de acesso²

Os modificadores de acesso são palavras chaves utilizadas para modificar o tipo de acesso a determinada variável.

const: Uma variável `const` é uma variável constante, que nunca muda. Portanto seus valores são atribuídos uma única vez, e não podem aparecer a esquerda do sinal de igual. A palavra chave `const` foi criada para substituir as instruções `# define`, a vantagem é que com `const` existe verificação de tipo, o que não ocorre com `define` (`const` só existe em C++).

Listing B.1: Modificadores de acesso.

```

//inclue a biblioteca de objetos relacionados a iostream
#include <iostream>

//estou usando o objeto cout, do namespace std
using std::cout;
using std::endl;

#include <iomanip>
using std::setprecision;

//criação de objeto constante (não pode mudar)
//global (visível em todo programa)
const double PI = 3.14159265358979;

int main()
{
    //criação de objeto constante (não pode mudar)
    //local (visível dentro de main)
    const float PI = static_cast< float >( ::PI );

    cout << setprecision( 20 ) << "Conteúdo de PI local = " << PI << endl;
    cout << setprecision( 20 ) << "Conteúdo de PI global = " << ::PI << endl;

    return 0;
}

/*
Novidade:
-----
-Declaração e definição de objeto constante
-Uso do operador de resolução de escopo (::) para obter uma variável do nível
anterior.

```

```

-Observe que PI global é double (+precisão) e PI local é float (-precisão)
float    7 dígitos de precisão
double  14 dígitos de precisão

-Uso do operador static_cast< >, static cast é usado para fazer conversões
em tempo de compilação (estáticas). Dentro do <> o tipo a converter.
Assim:
float b=3.333;
int a;
  a = (int) b; //cast de C
  a = static_cast < int > (b); //cast de C++
*/

/*
Saída:
-----
[andre@mercurio Cap2-Sintaxe]$ ./a.out
  Conteúdo de PI local = 3.1415927410125732422
  Conteúdo de PI global = 3.1415926535897900074
*/

```

volatile: Variáveis que podem ter seu valor alterado de modo inesperado. Tem utilidade em dispositivos mapeadores de memória, multitarefa.

mutable: Quando você quer permitir que um atributo de uma classe/estrutura possa ser modificado mesmo que o objeto seja declarado como const.

define: Define substitui todas as ocorrências de string1 por string2.

Exemplo:

```

//Uso de # define
# define string1 string2
# define CM_FileSave 2515
# define volume 10

```

Exemplo:

```

//Uso de const
const int m = 9, x = 25;
const float taxa = 0.7;
//Uso de volatile
volatile int tempo;
//Uso de const volatile
const volatile int alfa;
struct Funcionário{
  char nome[30];
  mutable int contador;}
const Funcionário func = {"Pedro",0};
//cria objeto func constante
//Erro func é const

```

```
strcpy(func.nome,"joão da silva");  
//o acesso é permitido graças ao especificador mutable  
func.contador = 1;
```

const e volatile: No exemplo acima, `alfa` é uma constante, como tem o `volatile` o compilador não faz qualquer suposição sobre os valores de `alfa`, um programa externo pode mudar o valor de `alfa`. Existem outros modificadores como `pascal`, `cdecl`, `near`, `far` e `huge`. Estes modificadores são utilizados para interfecear programas em C++ com programas em C, `pascal`, e para aumentar a portabilidade dos programas.

pascal: No C a chamada de uma função é substituída por uma sublinha seguida do nome da função, além de declarar a função como externa. Entretanto, se a função a ser chamada foi desenvolvida em `pascal` você deve declarar a função como externa e incluir a palavra chave `pascal`.

```
Exemplo:  
int pascal função(parâmetros); //Declara  
..  
int pascal função(parâmetros) //Define  
{  
}
```

cdecl: A palavra chave `cdecl` é usada para manter a sensibilidade as maiúsculas e minúsculas e a sublinha inicial usadas em C.

near, far, huge: As palavras chaves `near`, `far` e `huge` se aplicam a ponteiros. `near` força o ponteiro a ser de 2 bytes, `far` e `huge` forçam o ponteiro a ser de 4 bytes. `far` e `huge` diferem na forma do endereço.

```
Exemplo:  
//ponteiro p/ int  
near int* ptr_int;  
//ponteiro p/ double  
far double* ptr_double;  
//ponteiro p/char  
huge char* ptr_char;
```

B.3 Escopo das variáveis²

O primeiro conceito sobre declaração de variáveis é que um programa é dividido em níveis, para melhorar sua organização. Desta forma quando declaramos uma variável em determinado ponto do programa estamos fazendo com que está variável seja visível (pode ser acessada) no nível em que foi declarada e nos níveis abaixo. Lembre-se que uma variável só existe a partir de sua declaração.

O escopo de uma variável define onde ela pode ser utilizada. Existem 4 tipos de escopos: local, de função, de arquivo e de classe.

Local (de bloco): Quando uma variável é declarada dentro de um bloco ela é visível dentro deste bloco a partir de sua declaração, e nos níveis mais baixos.

De função: Quando uma variável é declarada dentro de uma função, ela só é visível dentro da função. Se a variável for declarada como `static` a mesma é criada uma única vez, mantendo-se viva mesmo depois do encerramento da função. Se a variável não for `static` a mesma deixa de existir quando a função termina.

Global: Quando declaramos uma variável fora de todas as classes, funções globais e blocos, então ela é uma variável global. Uma variável global pode ser vista por todos os arquivos que compõem o programa a partir de sua declaração.

Exemplo de variáveis que só são vistas no arquivo em que foram declaradas:

- Uma função inline declarada explicitamente.
- Uma variável `const`.
- Uma variável global estática.

Em resumo, se for inline, ou for `const` terá escopo de arquivo, caso contrário será visível por todos os arquivos.

De classe: Quando declarada dentro de uma classe, se for privada vai ser visível somente na classe em que foi declarada, se for `protected` vai ser visível na classe em que foi declarada e nas classes filhas e se for `public` em todo o programa.

static: Uma variável ou função `static` só é vista no arquivo em que foi declarada.

Exemplo:

```
//explicita define o seu valor
static int valor1 = 16;
//implícita, assume o valor 0.
static int valor2;
//visível somente neste arquivo
static int soma(int x, int y);
```

extern: O uso de `extern`, permite a compilação separada dos diversos arquivos que compõem um programa. Você deve usar `extern` na declaração de uma função ou procedimento que é definida em outros arquivos. O arquivo com a declaração `extern` aceita a existência da função de modo que você pode compilar as funções separadamente.

Exemplo:

```
//No arquivo A:
//cria a variável x (declara e define)
```

```
int x;
//No arquivo B:
//informa que x já existe (só declara)
extern int x;
```

Com a declaração `extern`, o arquivo B aceita a existência da variável `int x`; mas não aloca espaço de armazenamento para a mesma. Deste modo você pode compilar os arquivos A e B separadamente.

Listing B.2: Função e escopo - e14-escopo-a.cpp.

```
//-----Arquivo exemploA.cpp
//diz que existe a funcao1
extern void funcao1 ();

//Cria variável global, com nome x1
int x1 = 5;

//visível neste arquivo, permanente, global
static int x3 = 7;

int main ()
{
//visível dentro de main,
  int x1 = 7;

//visível dentro de main(), temporária
  float y1 = 44;
  for (int i = 0; i < 5; i++)
  {
    //visível dentro do for, temporária
    float z1, x1;

    /*existem três x1, um global, um de main, e um do for. Este é
      do for */
    x1 = 3;

    //z1=3
    z1 = x1;

    //z1=7, :: acessa x1 de dentro de main
    z1 = ::x1;

    //z1=5, acessa x1 global
    //z1=:: (::x1);
    //z1=44
    z1 = y1;
  }
//chama funcao1
funcao1 ();
}
```

Listing B.3: Função e escopo -e14-escopo-a.cpp.

```
//Arquivo -----exemploB.cpp
//Cria x2 global, permanente
```

```
int x2;

//Já existe um x1 do tipo int
extern int x1;

//x2=x1; //x2=5 não aceita

//declara função1
void funcao1 ();

//definição da função1
void funcao1 ()
{
//visível dentro de função1, temporário
float y2 = x2;

for (int i = 0; i < 5; i++)
{
//r4 tem escopo de bloco, temporário
int r4 = 3;

//r3 tem escopo de função, permanente
static int r3;

r3++;
x2 = r3 * i;
}

//r3 ainda existe é estático, só é inicializado na 1 vez
}
```

B.4 Sentenças para classes de armazenamento, escopo e modificadores de acesso

- Um objeto criado dentro de uma função ou bloco é eliminado quando a função ou o bloco termina, com exceção dos objetos estáticos.
- Só use static dentro de funções.
- A função abort() aborta o programa sem destruir os objetos estáticos.
- Em C++ todo objeto const deve ser inicializado na sua declaração.
- Evite usar # define, use const de C++.
- Qualquer uso de um nome deve ser não ambíguo em seu escopo. Depois de analisada a não ambiguidade é que será considerada a acessibilidade da variável. Quando digo que uma variável não deve ser ambígua em seu escopo, estou querendo dizer que o compilador ao passar por uma variável, deve ter certeza de qual variável você quer acessar, ele não pode ficar em dúvida.

Apêndice C

Operadores

Neste capítulo vamos apresentar a ordem de precedência dos operadores de C++. Quais os operadores de C++.

C.1 Introdução aos operadores

Os operadores foram definidos pela linguagem C++ e podem ser utilizados para realizar a comparação entre dois objetos

Os operadores estão divididos em operadores aritméticos, de atribuição e composto, de resolução de escopo, condicional, virgula, relacionais, incremento, decremento, deslocamento, new, delete, typedef, sizeof e de bits.

Apresenta-se na tabela C.1 a precedência dos operadores. O operador de mais alta prioridade é o operador de resolução de escopo (::), a seguir os parênteses () e depois os colchetes []. O operador de mais baixa prioridade é o operador virgula.

Tabela C.1: Precedência dos operadores.

1	2	3	4	5	6	7	8
::	dynamic_cast	!	delete	-	!=	+=	<<=
()	static_cast	~	delete[]	<<	& and	-=	>>=
[]	reinterpret_cast	(tipo)	.*	>>	^ or	*=	,
.	const_cast	sizeof	->*	<	ou	/=	
->	++i	&	*	<=	&& e	%=	
i++	--i	*	/	>	ou	&=	
i-		new	%	>=	?:	^=	
typeid		new[]	+	==	=	=	

Dica: Se precisar use parênteses para deixar o código mais claro.

C.2 Operadores de uso geral

C.2.1 Operadores aritméticos (+, -, *, /, %)

Símbolo Descrição

+	Soma
-	Subtração
*	Multiplificação
/	Divisão
%	Resto da divisão

Exemplo:

```
A = B + C;  
A = 8 - 3;  
B = 3 *( A / B);
```

C.2.2 Operadores de atribuição (=)

Exemplo:

```
A = B;  
A = B = C = D = 0;
```

C.2.3 Operadores compostos (+=, -=, *=, /=)

Os operadores compostos aumentam a eficiência dos programas pois só acessam o objeto uma única vez.

Exemplo:

```
A += B; //o mesmo que A = A + B;  
A -= B; //o mesmo que A = A - B;  
A /= B; //o mesmo que A = A / B;  
A *= B; //o mesmo que A = A * B;
```

C.2.4 Operadores relacionais (>, >=, <, <=, ==, !=)

Operadores:

>	maior
>=	maior ou igual
<	menor
<=	menor ou igual
==	igual
!=	diferente

C.2.5 Operadores lógicos (&&, ||, !, ==, !=)

Operador Símbolo Descrição:

AND	& & e
OR	ou
NO	! não, inverte o resultado
Igual	== testa a igualdade
Diferente	!= testa se é diferente

Exemplo:

```
if(x>y || x>z) return(z*y);
```

C.2.6 Operador condicional (?)

É o único operador ternário do C++. Se a expressão for verdadeira executa a expressão verdadeira. Veja o protótipo e o exemplo.

Protótipo:

(condição)? (expressão verdadeira): (expressão falsa);

Exemplo:

```
int maior = x > y ? x : y;
```

C.2.7 Operador incremento (++) e decremento (--)

Se você tem uma variável inteira e quer a incrementar de 1 (somar a 1) use o operador de incremento ++, para diminuir de 1, use o operador decremento --.

Para obter:

```
valor = valor + 1;
```

Faça:

```
valor++;
```

Para obter:

```
valor = valor - 1;
```

Faça:

```
valor-- --;
```

A operação de incremento pode ser do tipo postfix ou prefix.

Na operação postfix (i++) primeiro calcula e depois incrementa o valor.

Exemplo:

```
i++;
```

Na operação prefix (`++i`) primeiro incrementa e depois usa o valor.

Exemplo:

```
++i;
```

Exemplo:

```
int y = 5;           //Define y = 5
int x = 5 + y++;    //x = 10, y = 6
int z = 5 * ++y;    //y = 7, z = 5*7 = 35
```

C.2.8 Operador vírgula (a,b)

Avalia duas expressões onde a sintaxe só permite uma.

Prototipo: *(expressão da esquerda),(expressão da direita)*

Exemplo:

```
int x,y,z;
for(int min = 0,int max = 10; min < max; min++, max--)
{ cout << min << " " << max;}
```

Observe dentro do for a declaração de dois inteiros usando o separador virgula.

```
int min = 0,int max = 10
```

Os operadores new e delete são discutidos na seção sobre memória.

C.2.9 Operador módulo (%)

Veja no exemplo abaixo o uso da função rand da biblioteca matemática padrão de C e do operador módulo.

Exemplo:

```
#include <math>
int a = 7, b = 4;
int x = a% b;           //x = 3, é o resto da divisão
x = rand% 6            //x = resultado entre 0 e 5
int n = a+rand% (b+1); //número randomico entre a e b
```

C.3 Operadores de uso específico

C.3.1 Operador typedef

Com o uso da palavra chave typedef, pode-se atribuir um outro nome para um tipo definido (do sistema ou do usuário), ou seja, typedef pode ser usado para apelidar um tipo.

Apelida o tipo float de racional

```
typedef float racional;
```

Cria objeto PI do tipo racional

```
racional PI = 3.14159;
```

C.3.2 Operador sizeof e size_t

O operador sizeof retorna o tamanho do objeto em bytes. Quando aplicado a um vetor, retorna o número total de bytes do vetor. Pode ser aplicado ao ponteiro de uma função, mas não se aplica a uma função.

O operador size_t retorna o tipo.

```
int x, tipo;
tipo = size_t(x);
cout<<"0 tamanho em bytes de x é " << sizeof(x)<< endl;
```

C.3.3 Operador de resolução de escopo (::)

O operador de resolução de escopo é utilizado para identificarmos (e para o compilador identificar), qual variável estamos acessando. Se utilizarmos o operador de resolução de escopo dentro de uma função, estaremos acessando a variável externa a função.

```
int x = 70;
main(){
int x = 56;
int y = x; // y = 56
int z = ::x; // z = 70 pega o x externo a função main
}
```

C.3.4 Sentenças para operadores

- Performance: Em testes de comparação, sempre teste em primeiro lugar as condições mais prováveis.
- Bug: é um erro comum usar = para comparar dois valores, como na expressão if(x=3); quando o correto é if(x==3)
- BUG: Para inverter um bool use !.
- BUG: É um erro comum usar != no lugar de !=.

- Dica: não confunda um operador sobre bits (& e |) com operadores lógicos (&& e ||). Dica: bool pode assumir true(!=0) ou false(==0), assim 1 é true, 5 é true, 0 é false.

Listing C.1: Operadores de comparação.

```
#include <iostream>

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

int main()
{
    int a, b;
    cout << "Entre com dois numeros inteiros (a espaço b enter): ";
    //Observe abaixo a leitura de duas variaveis
    //em uma unica linha. Isto deve ser evitado.
    cin >> a >> b;
    cin.get();    //pega o enter

    if ( a == b )
        cout << a << "==" << b << endl;

    if ( a != b )
        cout << a << "!=" << b << endl;

    if ( a < b )
        cout << a << "<" << b << endl;

    if ( a > b )
        cout << a << ">" << b << endl;

    if ( a <= b )
        cout << a << "<=" << b << endl;

    if ( a >= b )
        cout << a << ">=" << b << endl;

    return 0;
}
/*
Novidade:
-----
-Uso de entrada de duas variáveis na mesma linha
    cin >> a >> b;

-Uso de cin.get() para retirar o enter o teclado.

-Uso dos operadores de comparação

-Uso do operador de controle if
*/
```

Listing C.2: Uso de sizeof.

```
//Tamanho dos deferentes tipos de C++
```

```
//determinados com o operador sizeof
#include <iostream>
using std::cout;
using std::endl;

#include <iomanip>

int main()
{
    char c;
    short s;
    int i;
    long l;
    float f;
    double d;
    long double ld;

    cout << "sizeof_c=" << sizeof c << endl;
    cout << "sizeof(char)=" << sizeof( char ) << endl;
    cout << "sizeof_s=" << sizeof s << endl;
    cout << "sizeof(short)=" << sizeof( short ) << endl;
    cout << "sizeof_i=" << sizeof i << endl;
    cout << "sizeof(int)=" << sizeof( int ) << endl;
    cout << "sizeof_l=" << sizeof l << endl;
    cout << "sizeof(long)=" << sizeof( long ) << endl;
    cout << "sizeof_f=" << sizeof f << endl;
    cout << "sizeof(float)=" << sizeof( float ) << endl;
    cout << "sizeof_d=" << sizeof d << endl;
    cout << "sizeof(double)=" << sizeof( double ) << endl;
    cout << "sizeof_ld=" << sizeof ld << endl;
    cout << "sizeof(longdouble)=" << sizeof( long double ) << endl;
    return 0;
}
/*
Novidade:
-Verificação do tamanho de cada tipo com sizeof
*/
.
```


Apêndice D

Controle

Apresenta-se neste capítulo, de forma resumida, o protótipo e exemplos de uso das estruturas de controle de C++. No final do capítulo são listados alguns programas que mostram o uso prático das estruturas de controle.

As instruções de controle são utilizadas em loopings, na realização de contagens, na escolha entre diferentes blocos de instruções.

No final do capítulo são apresentados alguns exemplos.

D.1 if

Se a expressão for verdadeira executa a próxima linha (ou bloco).

Protótipo:

```
if(expressão)
    ação;
```

Protótipo: *if(expressão)*

```
{ação1;ação2;...}
```

Exemplo:

```
int x=0, y=1;
if(x > y)
    cout << "x>y.";
```

D.2 if.....else

Se a expressão for verdadeira executa a ação1, se falsa a ação2.

Protótipo:

```
if(expressão1)
    ação1;
else
    ação2;
```

Exemplo:

```
int x=2; int y=3;
if( x >= y )
{
    cout<<'x>=y'<<endl;
}
else if(x<=z)
{
    cout<<'x<z'<<endl;
}
```

D.3 if.....else if.....else if.....else

O uso de if..else..if..else, produz um código rápido. Sempre coloque no início dos if..else os itens de maior probabilidade de ocorrência. O primeiro if que for verdadeiro vai ser executado os demais são pulados.

Cada else esta associado ao if imediatamente acima.

Protótipo:

```
if(expressão1)
ação1;
else if (expressão2)
ação2;
else if (expressão3)
ação3;
```

D.4 switch....case

A instrução *switch case* é utilizada quando se deseja fazer uma seleção entre uma série de opções. Se nenhuma opção satisfazer a condição vai realizar a instrução default.

Protótipo:

```
//resposta deve ser inteiro ou ser convertido para inteiro.
switch (resposta)
{
//se resposta=='a'
case 'a':
//realiza instruções 1
intruções1;
break;
//encerra o switch
case 'b':
intruções2;
break;
//opcional (sempre a última)
```

```

    default: instrução_default;
}

```

D.5 expressão? ação_verdadeira : ação_falsa;

O operador `?` é o único operador ternário do C++. Se a expressão for verdadeira realiza a ação_verdadeira, se a expressão for falsa realiza a ação_falsa.

Protótipo:

```

expressão? ação_verdadeira : ação_falsa;

```

Exemplo:

```

//se x < y retorna x, caso contrário y.
int z = x < y ? (x) : (y);

```

D.6 for(início;teste;incremento) ação;

O for é utilizado para a realização de um loop.

C++ permite declarar e inicializar o contador dentro do for. Os objetos declarados dentro do for só são visíveis dentro do for.

Dentro do for (), pode-se inicializar uma variável de controle. O teste é a condição a ser satisfeita, se verdadeiro continua executando o for, se falso encerra o for. A ação1 geralmente é uma ação de incremento/decremento realizada sobre a variável de controle.

Protótipo:

```

for(inicialização; teste; ação1)
ação2;

```

Exemplo:

```

for (int cont=3; cont< 20; cont++)
{
cout<<"\nO valor de cont é agora = "<< cont ;
}
cout << "\nfor encerrado"<<endl;

```

No exemplo acima, cria uma variável cont do tipo int e inicializa com o valor 3. A seguir testa se cont é menor que 20. Depois do teste, entra dentro do bloco e executa a linha (cout<<"\nO valor de cont é agora = "<< cont ;). Depois de realizada a execução do bloco, a ação1 é executada, isto é, cont++. E então o teste é novamente realizado.

Veja abaixo a sequência de execução:

```

int cont = 3;
cont < 20 ?
cout<<"\nO valor de cont é agora = "<< cont;
cont++;
cont <20 ?
cout<<"\nO valor de cont é agora = "<< cont;

```

```

cont++;
cont < 20 ?
cout << "\nO valor de cont é agora = " << cont;
cont++;
....
//quando cont = 20 o for é encerrado
cout << "\nfor encerrado" << endl;

```

D.7 while (teste){instrução};

Enquanto o teste for verdadeiro executa a instrução. O comando while pode ser usado no lugar de for, veja o protótipo e exemplo a seguir.

Protótipo:

```

while (TESTE)
{INSTRUÇÃO};

```

Exemplo:

```

//0 for abaixo,
for(e1;e2;e3)
{
    ação;
};
//pode ser realizado usando while
//desta forma
e1;
while(e2)
{
    ação;
    e3;
}

```

D.8 do {ação} while (teste);

O comando do while, realiza a ação e somente depois testa.

Protótipo:

```

do
{ação;}
while (teste); //se satisfazer o teste vai repetir.

```

Dica: Evite o uso de do...while();.

D.9 break

Quando um `break` é encontrado, o bloco é encerrado. Pode ser usado dentro de um `for`, em um `while`, `do...while`, ou `switch`. No exemplo a seguir pula para próxima instrução fora do bloco.

```
Exemplo:
for(int i=0; i<10 ;i++)
{
ação1;
if(condição) break;
ação2;
}
/*Neste exemplo, se a condição for verdadeira a
ação2 não será executada, e o for é encerrado*/
```

D.10 continue

O controle `continue` (usado em `for`, `while`, `do..while`), faz com que o programa prossiga na expressão que controla o loop. Quando `continue` é encontrado todos os próximos passos não são executados pulando para o próximo passo do loop. Se assemelha a um comando `goto`.

Enquanto `break` encerra o loop, `continue` continua sem executar as linhas que estão abaixo.

```
Exemplo:
for(int i=0;i<10;i++)
{ação1;
if(condição) continue;
ação2;
}
/*
Se a condição for verdadeira, a ação2 não é executada, o
próximo passo a ser executado é o comando for*/
```

Apresenta-se a seguir um conjunto de programas com exemplos de estruturas de controle. Lembre-se, você deve digitar estes programas para aprender na prática a usar a sintaxe de C++, isto é, evite compilar a versão baixada na internet.

Listing D.1: Uso de `for`.

```
#include <iostream>

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

int main()
{
    //for( inicialização; condição de saída; ação apos cada loop)
    //{.....realizar.....}

    for ( int i = 1; i <= 10; i++ )
```

```

        cout << i << endl;

    return 0;
}

/*
Novidade:
-----
-Uso de for para realizar um looping.
*/

```

Listing D.2: Uso de for encadeado.

```

#include <iostream>

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

int main()
{
    int x, y;

    cout << "Entre com dois números inteiros positivos (a espaço b enter): ";
    cin >> x >> y;
    cin.get();

    for ( int i = 1; i <= y; i++ )
    {
        for ( int j = 1; j <= x; j=j+1 )
            cout << '.';

        cout << endl;
    }

    return 0;
}

/*
Novidade:
-----
-Uso de for encadeado.
*/

```

Listing D.3: Uso de while.

```

#include <iostream>
using std::cout;
using std::endl;

int main()
{
    int i = 1;

    //Protótipo:
    //do{ realizar }

```

```

//while(condição);
//não esqueça o ;ao final do while

//sempre realiza o looping pelo menos 1 vez.
do
{
    cout << i << "░░";
}
while ( ++i <= 10 );

cout << endl;

return 0;
}
/*
Novidade:
-----
_Uso de do{} while();
*/

```

Listing D.4: Uso de switch.

```

#include <iostream>
using std::cout;
using std::cin;
using std::endl;

int main()
{
    int caracter,          //A vírgula pode ser usada para
        a = 0,            //separar campos
        e = 0,
        i = 0,
        o = 0,
        u = 0;

    //Observe o formato do uso do cout
    cout << "Digite uma vogal ░░a,e,i,o,u:" <<endl
        << "Para encerrar, digite o caracter de fim de arquivo." <<endl
        << "No windows/DOS (ctrl+z) no unix/linux (ctrl+d):\n" << endl;

    //cin.get le um caracter do teclado e armazena em caracter.
    //Depois compara com EOF que significa end of file
    //ou seja, se chegou ao final do arquivo ou se
    //o usuario digitou a sequencia para encerrar a entrada (ctrl+z)
    while ( ( caracter = cin.get() ) != EOF )
    {

        switch ( caracter )
        {

            case 'A':
            case 'a':
                ++a;
                break;

```

```

    case 'E':
    case 'e':
        ++e;
        break;

    case 'I':
    case 'i':
        ++i;
        break;

    case 'O':
    case 'o':
        ++o;
        break;

    case 'U':
    case 'u':
        ++u;
        break;

    case '\n':
    case '\t':
    case '\_':
        break;

    default:
        cout << "Entrou na opção default do switch, ou seja, \n";
        cout << "voce entrou com uma letra errada." << endl;
        cout << "Repita a operação." << endl;
        //o break abaixo é opcional
        break;
}
}

cout << "\n\nTotal para cada letra:"
<< "\nA: \n" << a
<< "\nE: \n" << e
<< "\nI: \n" << i
<< "\nO: \n" << o
<< "\nU: \n" << u << endl;

return 0;
}
/*
Novidade:
-----
-Uso de swith(var) {case op:...break; ...default: ...break;}
-Uso de ctrl+z (no windows) para encerrar uma entrada de dados.
*/

/*
Saída:
-----
[andre@mercurio Cap2-Sintaxe]$ ./a.out

```



```

Digite uma vogal a,e,i,o,u:
Para encerrar, digite o caracter de fim de arquivo.
No windows/DOS (ctrl+z) no unix/linux (ctrl+d):

a a e e i o u a a e i o u u

Total para cada letra:
A: 4
E: 3
I: 2
O: 2
U: 3

*/

```

Listing D.5: Uso de break.

```

#include <iostream>
using std::cout;
using std::endl;

int main()
{
    int x;
    for ( x = 1; x <= 10; x++ )
    {
        if ( x == 5 )
            break;    //Encerra o looping quando x==5

        cout << x << '␣';
    }

    cout << "\nSaiu␣do␣looping." << endl;
    return 0;
}
/*
Novidade:
-----
-Uso de break para encerrar um looping.
*/

/*
Saída:
-----
[andre@mercurio Cap2-Sintaxe]$ ./a.out
1 2 3 4
Saiu do looping.
*/

```

Listing D.6: Uso de continue.

```

#include <iostream>

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

```

```

int main()
{
    for ( int x = 1; x <= 20; x+=1 )
    {
        if ( x == 5 )
            continue; // Pula para próximo passo do looping

        cout << x << ";";
    }

    cout << "\nUm continue continua o looping, mas pula todas as linhas abaixo."
        ;
    cout << "\nObserve acima que não imprimiu o número 5." << endl;
    return 0;
}

/*
Novidade:
-----
-Uso de continue
*/

/*
Saída:
-----
[andre@mercurio Cap2-Sintaxe]$ ./a.out
1;2;3;4;6;7;8;9;10;11;12;13;14;15;16;17;18;19;20;
Um continue continua o looping, mas pula todas as linhas abaixo.
Observe acima que não imprimiu o número 5.
*/

```

Listing D.7: Uso do operador de incremento.

```

#include <iostream>
using std::cout;
using std::endl;

int main()
{
    int contador=0;
    cout << contador << endl;           // Escreve 0 na tela
    cout << (contador++) << endl;       // Escreve 0 e depois faz contador=1
    cout << contador << endl << endl;   // Escreve 1

    contador = 0;
    cout << contador << endl;           // Escreve 0
    cout << (++contador) << endl;       // Incrementa contador=1 e
                                        // depois escreve na tela
    cout << contador << endl;           // Escreve 1

    return 0;
}

```

```

/*
Novidade:
-----
-Uso do operador incremento

x++      -> Primeiro usa o valor de x e depois incrementa (pós fixada)
++x      -> Primeiro incrementa x e depois usa (pré fixada)
*/

```

Listing D.8: Uso do operador while, exemplo 1.

```

#include <iostream>

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

int main()
{
    int x, y ;

    cout << "Entre com a base (inteiro): ";
    cin >> x;

    cout << "Entre com o expoente (inteiro): ";
    cin >> y;
    cin.get();

    //Prototipo
    //while(condicao)
    //    {acao; };
    int i=1;
    double potencia = 1;
    while ( i <= y )
    {
        //    *=
        //é o mesmo que potencia = potencia*x; mas é mais rápido
        potencia *= x;
        ++i;
    }

    cout << potencia << endl;
    return 0;
}

```

```

/*
Novidade:
-----
-Uso de while
*/
/*
Saída:
-----
[andre@mercurio Cap2-Sintaxe]$ ./a.out

```

```

Entre com a base (inteiro): 3
Entre com o expoente (inteiro): 4
81
*/

```

Listing D.9: Uso do operador while, exemplo 2.

```

#include <iostream>
using std::cout;
using std::endl;

int main()
{
    int y, x = 1, total = 0;

    while ( x <= 10 )
    {
        y = x * x;
        cout << x << "\t" << y << endl;
        total += y;
        ++x;
    }

    cout << "Total=□" << total << endl;
    return 0;
}

/*
Saída:
-----
[andre@mercurio Cap2-Sintaxe]$ ./a.out
1      1
2      4
3      9
4     16
5     25
6     36
7     49
8     64
9     81
10    100
Total= 385
*/

```

Listing D.10: Uso do operador módulo e do operador ?.

```

#include <iostream>

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

int main()
{
    int contador= 1;

    /// é o operador módulo, retorna o resto da divisão.

```

```

//Assim 5%4 retorna 1
cout<<"5%4="<< (5%4) <<endl;

//Protótipo
//condição ? açãooverdadeira : açãoFalsa;

//o operador ? é escrito assim
//int z =      x > y ? x : y ;
//se x>y retorna x, senão retorna y.
int x = 5;
int y = 3;
int z =      x > y ? x : y ;

cout << "\nx=5,y=3,z=□x>y?□x:y;"<<endl;
cout << "z="<< z <<endl;

while ( contador <= 10 )
{
//se contador%2 for verdadeiro retorna ****, senao ++++++
//sempre que contador for par, a divisao retorna zero.

cout << (contador % 2 ? "****" : "++++++") << endl;
++contador;
}

return 0;
}

/*
Novidade:
-----
-Operador %, ou resto da divisão
-operador ?
*/

/*
Saída:
-----
[andre@mercurio Cap2-Sintaxe]$ ./a.out
5%4=1

x=5,y=3,z = x>y? x:y;
z=5
****
+++++++
****
+++++++
****
+++++++
****
+++++++
****
+++++++
****
+++++++
*/

```


Apêndice E

Funções - Parte II

Apresenta-se neste capítulo uma complementação dos conceitos apresentados no capítulo 10.

E.1 Uso de argumentos pré-definidos (inicializadores)

O uso de argumentos pré-definidos (inicializadores), consiste em atribuir valores aos parâmetros de uma função. Assim, quando a função é chamada sem argumentos, serão usados os argumentos pré-definidos. No exemplo abaixo a função `f` tem os parâmetros `a`, `b` e `c` previamente inicializados com os valores 4, 7 e 9.3, respectivamente.

```
Exemplo:
int f(int a=4, int b=7, float c=9.3)
{
    return a+b+c;
}
//A função pode ser chamada das seguintes formas:
f (77, 5, 66.6); //a=77, b=5, c=66.6
f(33, 75);      //a=33, b=75, c=9.3
f(67);         //a=67, b=7, c=9.3
f();           //a=4, b=7, c=9.3
```

A variável que deixa de ser fornecida é aquela que está mais à direita.

A declaração de argumentos default corresponde a uma sobrecarga da função realizada pelo compilador. Veja conceitos de sobrecarga de função e de operadores no capítulo sobre Sobrecarga.

E.2 A função `main()` e a entrada na linha de comando²

A função principal de um programa é chamada `main()`. Quando um programa é executado na linha de comando do DOS/Unix/Linux, o sistema operacional chama a função `main()` do programa e passa para o programa dois argumentos; O texto da linha de comando e o número de strings da linha de comando.

Assim, você pode usar a linha de comando para entrada de informações em seu programa, veja abaixo o protótipo da função `main`.

Protótipo: `main(int argc, char *argv[]) {.....}`
 sendo **argc** o número de elementos da linha de comando
 e **argv[]** o vetor para as strings digitadas na linha de comando

Exemplo:

```
//Na linha de comando do Linux digitou-se:
cp funcoes.lyx funcoes.lyx~
//O programa interpreta o seguinte:
//argc=3
//argv[0]="cp"
//argv[1]="funcoes.lyx"
//argv[2]="funcoes.lyx~"
```

No exemplo acima, o programa recebe um vetor com as strings {"cp", "funcoes.lyx", "funcoes.lyx~"} e o número de parâmetros que é 3.

E.3 Funções recursivas²

O exemplo abaixo apresenta uma função recursiva, que calcula o fatorial de um número.

Exemplo:

```
//Números de Fibonacci
long fibonacci(long n)
{
  if(n==0 || n==1)
    return n;
  else
    return fibonacci(n-1)+fibonacci(n-2); //recursão
}
```

Observe que em funções recursivas existe um excesso de chamadas de funções, o que diminui a performance do sistema. Apresenta-se a seguir um exemplo.

Listing E.1: Função recursiva.

```
#include <iostream>
using std::cout;
using std::endl;

#include <iomanip>
using std::setw;

//Declaração da função
long int fatorial( long int );

int main()
{
  long int min;
  cout << "Entre com o valor mínimo: ";
  cin >> min;
```



```

long int max;
cout << "Entre com o valor máximo (max): ";
cin >> max;
cin.get();           //pega o enter

for ( long int i = min; i <= max; i++ )
    cout << setw( 2 ) << i << "!=" << fatorial( i ) << endl;

return 0;
}

//Definição da função
//fatorial 5=5*4*3*2*1
long int fatorial( long int n )
{
    //se for menor ou igual a 1 retorna 1 (finaliza)
    if ( n <= 1 )
        return 1;

    //se for >1 chama fatorial novamente (recursão)
    else
        return n * fatorial( n - 1 );
}

/*
Novidade:
-----
-Uso de recursão na chamada de funções
-Só chamar funções recursivas em casos extremos, a chamada
de uma função consome tempo de processamento.

No exemplo acima a função fatorial, é chamada recursivamente.
Observe que se o valor de n for grande,
o tipo long int pode ter seu intervalo estourado.
Ou seja, o fatorial pode retornar um valor errado (negativo).
Neste caso você tem de trocar int por float.
*/

/*
Saída:
-----
[andre@mercurio Cap2-Sintaxe]$ ./a.out
Entre com o valor mínimo (min) : 5
Entre com o valor máximo (max) : 9
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
*/

```

E.4 Uso de elipse ... em funções³

A elipse se refere a três pontos como parâmetro de uma função. Uma elipse numa lista de parâmetros significa que a função pode receber mais parâmetros. Um exemplo é a função `printf` que pode receber um número variado de parâmetros. Para maiores informações, de uma olhada na função `printf` e no `help`.

Exemplo:

```
//Protótipo
double media(int i,...); //i é o número de elementos
//Definição
double media(int i,...)
{
double total = 0;
//Cria lista variavel
va_list variavel;
//Inicializa lista variável
va_start(variavel,i);
for( int j = 1; j <= i; j++ )
    total += va_arg(variavel,double);
//Encerra lista ap
va_end(variavel);
}
```

Observe que a função `va_start` inicializa a retirada de variáveis da pilha, a função `va_arg` retira uma variável da pilha especificando o tamanho em bytes da variável (o que é feito passando o `double`), e a função `va_end` finaliza a retirada de dados da pilha. Isto significa que o uso de elipse altera o estado da pilha e seu uso deve ser evitado, ou usado somente após a função ter sido extensivamente testada.

E.5 Sentenças para funções

- O nome de uma função é um ponteiro com endereço constante, definido na compilação.
- O retorno de uma função pode ser uma chamada a outra função ou um objeto.
- Evite usar parâmetros de funções com nome igual ao de outras variáveis, o objetivo é evitar ambiguidades.
- Em `c++` todas as funções precisam de protótipos. O protótipo auxilia o compilador a encontrar erros.
- Analise os parâmetros das funções. Se os mesmos não são alterados, devem ser declarados como `const`.
- Objetos grandes devem ser passados por referência ou por ponteiros.
- Em `c++` podemos declarar função assim:

- `void nome(); //antes void nome(void)`
- Quando for acessar funções de outras linguagens use
Exemplo:
`extern "C" retorno nomefunção(parâmetros);`
`extern "FORTRAN" ...; //e não Fortran`
`extern "Ada".....; //e não ADA`
- Quando utilizamos uma matriz como argumento de uma função, os valores são passados por referência e não por cópia. Se a matriz estiver dentro de uma estrutura ela será passada por cópia.
- Performance: Parâmetros passados por referência aumentam a eficiência pois os valores não são copiados.
- Na função main, o retorno do valor 0, informa ao sistema operacional, que o programa terminou com sucesso. Qualquer valor diferente de zero indica um erro.
- Dentro da função main inclua um mecanismo simples de tratamento de excessões. Veja capítulo de tratamento de excessões.
- Segurança: Se você quer ter certeza de que o parâmetro não vai ser alterado, deve passá-lo como referência constante, veja o exemplo abaixo.

Exemplo:

```
//0 especificador const informa que a variável
//é constante e não pode ser alterada dentro da função
//Deste modo a função pode acessar a variável
//mas não pode modificá-la.
funcao(const tipo& variável);
```

E.6 Exemplos

Apresenta-se a seguir algumas listagens com exemplos de funções.

Apresenta uma função simples, para calcular o cubo de um número.

Listing E.2: Função cubo.

```
#include <iostream>
using std::cout;
using std::endl;

//Abaixo declara uma função
//Protótipo: retorno_da_funcao Nome_da_funcao (parâmetros)
int cubo ( int y );

int main()
{
    int min , max ;

    cout<<"Entre com o intervalo (valor mínimo e máximo) (ex: 3 e 10) : ";
```

```

cin >> min >> max;
cin.get();

for ( int x = 1; x <= 10; x++ )
    cout << "cubo de ("<<x<<")="<< cubo(x) << endl;

return 0;
}

//Definição da função: retorno_da_funcao Nome_da_funcao (parâmetros)
int cubo( int y )
{
    return y * y * y;
}
/*
Dica:
-----
    Depois de cada cin >> x;
    coloque um cin.get()
    o cin.get() pega e joga fora o return que você digitou.
*/
/*
Novidade:
-----
-Declaração, definição e uso da função cubo.
*/

```

Apresenta uma função simples, sem retorno.

Listing E.3: Função com void.

```

#include <iostream>

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

void f();          //C++
void g( void );   //C precisa do segundo void

int main()
{
    cout<<"\a\n";
    cout<<"Executar f() ou g() ?";
    char resp='f';
    cin>>resp;
    cin.get();    //pega o enter

    if(resp=='f' || resp=='F')
        f();
    else if(resp=='g' || resp=='G')
        g();
    else
    {
        cout<<"Seleção errada, selecionou ("<<resp<<")"<<endl;
        main(); //recursão
    }
    return 0;
}

```

```

}

void f()
{
    cout << "Função void não retorna nada (selecionou (f))" << endl;
}

void g ( void )
{
    cout << "Função void não retorna nada (selecionou (g))" << endl;
}
/*
Novidade:
-----
Uso de função com void
*/
/*
Saída:
-----
[andre@mercurio Cap2-Sintaxe]$ ./a.out
Executar f() ou g() ?f
Função void não retorna nada (selecionou (f))
[andre@mercurio Cap2-Sintaxe]$ ./a.out

Executar f() ou g() ?g
Função void não retorna nada (selecionou (g))
[andre@mercurio Cap2-Sintaxe]$ ./a.out

Executar f() ou g() ?h
Seleção errada, selecionou (h)

Executar f() ou g() ?f
Função void não retorna nada (selecionou (f))
*/

```

Apresenta uma função em linha.

Listing E.4: Função em linha (volume esfera).

```

#include <iostream>
using std::cout;
using std::cin;
using std::endl;

//Variáveis constantes devem ser declaradas com maiúsculas
const double PI = 3.14159;

/*
Um programador C antigo usaria
#define PI 3.14159
*/

//Função em linha
inline double VolumeEsfera( const double raio )
{
    return 4.0 / 3.0 * PI * raio * raio * raio;
}

```

```

/*
Um programador C antigo usaria uma macro,
sem nenhuma verificação de tipo.
#define Volume(raio) (4.0/3.0 * PI * raio * raio * raio)
*/

int main()
{
    double raio;

    cout << "Entre com o raio: ";
    cin >> raio; cin.get();

    cout << "A esfera de r=" << raio << " tem volume V=" << VolumeEsfera( raio )
        << endl; //1
//cout << "A esfera de r=" << raio << " tem volume V=" << VolumeEsfera( raio ) <<
    endl; //2
//cout << "A esfera de r=" << raio << " tem volume V=" << VolumeEsfera( raio ) <<
    endl; //3
//cout << "A esfera de r=" << raio << " tem volume V=" << VolumeEsfera( raio ) <<
    endl; //4
//cout << "A esfera de r=" << raio << " tem volume V=" << VolumeEsfera( raio ) <<
    endl; //5

    return 0;
}

```

```

/*
Novidade:
-----
-Uso de função com parâmetro const
-Uso de função inline, C++ usa funcoes inline no lugar de macros.
-Lembre-se que C++ usa const no lugar de #define, a vantagem é verificar
o tipo do objeto que esta sendo passado para a funcao.

```

```

Trabalho:
1-Verifique o tamanho do seu programa em bytes.
(aprox 2215 bytes no linux)
2-Tire os comentários //
3-Recompile o programa e então verifique o tamanho do seu programa em bytes.
(2218 bytes no linux)

```

O tamanho aumentou, porque uma funcao inline é colada onde esta sendo chamada, deixando o programa maior e mais rápido.

```
*/
```

```

/*
Saída:
-----
[andre@mercurio Cap2-Sintaxe]$ ./a.out
Entre com o raio: 5
A esfera de r=5 tem volume V= 523.598
*/

```

Apresenta uma outra função em linha.

Listing E.5: Função em linha exemplo 2.

```
#include <iostream>

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

/*Declaração da função em linha*/
inline double cubo( const double lado ) { return lado * lado * lado; }

int main()
{
    cout << "Entre com a dimensão do cubo: ";

    double dim;
    cin >> dim; cin.get();
    cout << "Volume do cubo " << dim << " é " << cubo( dim ) << endl;
    return 0;
}

/*
Dica:
-----
-Quando tiver funções pequenas, que são
chamadas diversas vezes, por exemplo dentro
de um for, especifique a mesma como inline
para aumentar o desempenho do programa
*/
/*
Novidade:
-----
-Uso de funções em linha.
*/
/*
Saída:
-----
[andre@mercurio Cap2-Sintaxe]$ ./a.out
Entre com a dimensão do cubo: 5
Volume do cubo 5 é 125
*/
```

Apresenta uma função utilizada para gerar números randômicos.

Listing E.6: Exemplo de uso da biblioteca <cstdlib>.

```
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

#include <iomanip>
using std::setw;

//Fornece a função srand e rand (número randomico de C)
//A biblioteca abaixo
//no C chama stdlib.h
//no C++ chama cstdlib
```

```
#include <cstdlib>

int main()
{
    //semente do gerador de número randômico
    int semente;
    cout << "Entre com uma semente: ";
    cin >> semente;
    cin.get();

    int min;
    cout << "Entre com o valor mínimo (a): ";
    cin >> min;
    cin.get();

    int max;
    cout << "Entre com o valor máximo (b): ";
    cin >> max;
    cin.get();

    //Passa a semente para o gerador de números aleatórios
    srand( semente );

    //Chama 50 vezes a função rand
    for ( int i = 1; i <= 50; i++ )
    {
        cout << setw( 10 ) << ( min + rand() % (max - min +1) ) ;

        if ( i % 5 == 0 )
            cout << endl;
    }

    return 0;
}
```

```
/*
Dica:
-----
-Use a função módulo (%) para obter o resto de uma divisão
-Use ( min + rand() % (max - min +1) )
para obter um número randômico entre min e max
-Use a função setw(10),
para setar a largura do campo de saída para 10 caracteres.
*/
```

```
/*
Novidade:
-----
-Use de função da biblioteca padrão de C <cstdlib>
A função rand e srand.
*/
```

```
/*
Exercício:
-----
Use o find de seu computador para localizar
a biblioteca cstdlib.
*/
```


Dê uma olhada rápida no arquivo `cstdlib` e localize a declaração das funções `srand` e `rand`

**/*

*/**

Saída:

[andre@mercurio Cap2-Sintaxe]\$./a.out

Entre com uma semente : 4

Entre com o valor mínimo (a) : 1

Entre com o valor máximo (b) : 10

2	4	5	7	4
8	6	4	9	6
5	5	5	5	4
5	3	2	2	10
8	5	9	10	4
9	3	3	1	1
1	4	5	7	10
10	5	5	4	3
2	10	7	6	5
2	3	9	3	4

**/*

Apêndice F

Ponteiros - Parte II

Apresenta-se neste capítulo conceitos adicionais sobre ponteiros. Os ponteiros foram apresentados no capítulo 12.

F.1 Operações com ponteiros (+/-)²

Operações que podem ser realizadas com ponteiros:

- Subtrair dois ponteiros. Retorna o número de elementos entre os dois ponteiros.
- Pode-se comparar ponteiros com (> , >=, <, <= , =).
- Pode-se incrementar (ptr++) e decrementar (ptr-) um ponteiro. O valor numérico do incremento é definido pelo valor retornado p/ SIZEOF.

Exemplo:

```
//n=número de variáveis entre ptr1 e ptr2
n = ptr1 - ptr2;
ptr++ ;
//incrementa ptr,
//se ptr aponta para o elemento 5 de
//uma matriz, ptr++, faz com que passe
//a apontar para o elemento 6 da matriz.
ptr-- ;
//decrementa ptr
Ptr = ptr + 4 ;
//4 significa 4 variáveis a frente, e não 4 bytes.
//Se apontava para o 3° elemento
//passa a apontar para o 7° elemento.
```

F.2 Ponteiro void²

Um ponteiro void é um ponteiro de propósito geral, que aponta para qualquer tipo de objeto e é usualmente usado como parâmetro de funções de propósito geral.

Um ponteiro void deve ser convertido para um tipo com tamanho conhecido em tempo de compilação, pois um ponteiro void não tem tamanho.

Veja, se um ponteiro é do tipo `int*`, a variável apontada tem o tamanho de um inteiro, assim, o compilador sabe quantos bits deve ler a partir do endereço especificado pelo ponteiro. Quando um ponteiro é do tipo void, o compilador só sabe a posição para a qual ele aponta, mas não sabe o quantidade de bits que devem ser lidos.

Exemplo:

```
void* ptr_void; //ptr é um ponteiro para qualquer coisa.
int * ptr_int; //ponteiro para int
int i=5;       //cria int i
ptr_int = & i; //endereço de i em ptr_int
ptr_void = & i; //ou ptr_void = ptr_int;
*ptr_int =5;   //coloca 5 em i
*ptr_void = 7; /*ERRO, void não pode usar
                operador localização *. */
/*é necessário formatar o ponteiro void, definindo o tamanho da
memória que vai ser acessada*/
*(int*) ptr_void = 7; //ok, converte ptr_void para int*
```

F.2.1 Sentenças para ponteiro void

- O ponteiro void não permite o uso do operador de localização(*) e endereçamento(&).
- O conteúdo de void só pode ser acessado depois de sua definição.
- Um ponteiro void pode ser igualado a outros ponteiros e vice versa.
- Observe o uso da mesma palavra chave para diferentes objetivos:
 - p/ lista de argumentos. void = nenhum argumento
 - p/ retorno de função. void = nenhum retorno
 - p/ ponteiros. void = qualquer tipo de dados.

F.3 Ponteiro para ponteiro³

Quando um ponteiro aponta para outro ponteiro. Neste caso o carteiro tem que andar bem mais, veja o exemplo.

Exemplo:

```
int variavel=33;
//ponteiro e endereçamento indireto simples
int* ptr1;
//ponteiro c/endereçamento indireto duplo
int **ptr2;
//ponteiro c/endereçamento indireto triplo
int***ptr3;
//coloca endereço da variável em ptr1, ou seja,
```

```
//ptr1 aponta para variável
ptr1=& variável;
//ptr2 aponta para ptr1
ptr2=& ptr1;
//ptr3 aponta para ptr2
ptr3=& ptr2;
//armazena 5 em variável (usando ptr1)
*ptr1=5;
//armazena 7 em variável (usando ptr2)
**ptr2=7;
//armazena 14 em variável (usando ptr3)
***ptr3=14;
/* Neste último caso
***ptr3=14;
o pobre do carteiro tem muito trabalho.
Ele pega o número 14 e vai até a casa de ptr3,
chegando lá recebe a ordem de ir até a casa de ptr2;
já cansado chega até a casa de ptr2,
e para sua surpresa recebe a ordem para ir até ptr1,
chateado e com fome, vai até a casa de ptr1;
para sua desgraça, recebe a ordem de ir até a casa
da variável, o pobre coitado, leva o número 14
até a casa da variável e como não é de ferro descansa.
Depois se filia a CUT e pede aumento de salário.
*/
```

Em resumo:

Quando você faz **tipo *ptr**; esta criando um ponteiro para o tipo.

Quando você usa **ptr1=& algo**; esta armazenando endereço de algo na variável ponteiro.

Quando você usa **int **ptr2=& ptr1**; esta criando ptr2 e armazenando o endereço de ptr1 em ptr2, ptr2 é um ponteiro de ponteiro.

Quando você usa **int **ptr22=ptr2**; ptr22 vai apontar para a mesma variável que ptr2.

Quando você usa **ptr=& x; *ptr=55**; esta armazenando 55 na variável apontada por ptr, no caso a variável x.

F.4 Ponteiro de Função³

Você pode criar um ponteiro para uma função.

Exemplo:

```
//protótipo da função
double cubo(double x);
//definição da função
double cubo(double x)
{
    return x*x*x;
}
```

```

}
//protótipo do ponteiro
double (*ptr_função)(double x);
//Armazena o endereço da função no ponteiro
ptr_função=& nome_função;
//Usa ponteiro p/executar a função
Resultado = *ptr_função(x);

```

F.5 Sentenças para ponteiros²

- Um ponteiro qualquer (desde que não seja const e volatile) pode ser convertido para void*.
- Um ponteiro pode ser convertido em qualquer tipo integral que seja suficientemente grande para contê-lo.
- Um ponteiro de um objeto de tamanho n pode ser convertido para ponteiro do objeto de tamanho m, se $n > m$.
- Sempre inicializar os ponteiros (ptr=NULL;), sempre deletar os ponteiros que usarem new.
- Um ponteiro para uma classe derivada pode ser convertido em ponteiro para a classe base.
- Uma referência a uma classe pode ser convertida numa referência a uma classe base acessível.
- Um ponteiro para o objeto B pode ser convertido em ponteiro para o objeto D, sendo B a classe básica de D, se a conversão for direta e não ambígua, e se B não é classe base virtual.

Listing F.1: Uso do operador de endereço e sizeof.

```

/*
-----
LMPT/NPC: Curso Interno- Sintaxe de C++
Autor: André Duarte Bueno
Arquivo: endereco-sizeof.cpp
-----
*/
#include<iostream.h>
void main()
{
char nome []="nucleo_de_pesquisa_em_construcao";
cout << "char_nome []=\\"nucleo_de_pesquisa_em_construcao\\"";
cout << "\n_nome=_ " << nome;
cout << "\n*_nome=_ " << *nome;
cout << "\n&nome=_ " << &nome;
cout << "\n(int)_nome=_ " << (int)nome;
cout << "\n(int)_*_nome=_ " << (int)*nome;
cout << "\n(int)_&nome=_ " << (int)&nome;
cout << "\n(int)_&nome[_]=_" << (int)&nome;
cout << "\n(int)_&nome[0]=_" << (int)&nome[0];
cout << "\n_nome=_ " << nome;
cout << "\nsizeof(nome)=_" << sizeof(nome);
cout << "\nsizeof(&nome)=_" << sizeof(&nome);

```

```

cout << "\nsizeof(&nome[0])_=_ " << sizeof(&nome[0]);
cout << "\nsize_t(nome)_=_ " << size_t(nome);
cout << "\nsize_t(&nome)_=_ " << size_t(&nome);
cout << "\nsize_t(&nome[0])_=_ " << size_t(&nome[0]);
cin.get();
}
/*
-----
Novidades:
-Uso de sizeof
-Uso do caracter \
-----
*/

```

Listing F.2: Uso de sizeof 1.

```

// O operador sizeof retorna o número de elementos do vetor
#include <iostream>
using std::cout;
using std::endl;

size_t dimensao( int * );
size_t dimensao( int *ptr )
{
    return sizeof( ptr );
}

int main()
{
    int v[ 20 ];

    cout << "O número de bytes do vetor v é de " << sizeof( v ) << endl;
    cout << "sizeof(int*ptr)= " << dimensao( v ) << endl;
    return 0;
}
.

```


Apêndice G

Estruturas, Uniões e Enumerações

G.1 Estrutura (struct)

Uma estrutura permite reunir um conjunto de objetos, dentro de uma entidade única.

Aconselha-se o uso de uma struct apenas para reunir objetos (de acordo com sua definição inicial) e deixar para as classes a reunião de objetos e métodos.

G.1.1 Definindo estruturas

O protótipo para definição de uma estrutura, inicia com a palavra chave *struct*, a seguir o nome da estrutura e o bloco com a definição dos objetos que fazem parte da estrutura.

Protótipo:

```
struct nome_estrutura
{
    tipo_1 variável1;
    tipo_2 variável2;
};
```

Depois de definida uma estrutura, o seu nome passa a ser um tipo do usuário, ou seja, a estrutura pode ser usada da mesma forma que qualquer outro tipo de C++. Veja a seguir como criar e usar uma struct de C.

G.1.2 Criando um objeto de uma estrutura

Pode-se criar um objeto de uma estrutura do mesmo modo que se cria um objeto qualquer de C++.

Protótipo:

```
nome_estrutura nome_objeto;
```

Pode-se criar um vetor de estruturas.

Protótipo:

```
nome_estrutura v[n];
```

G.1.3 Acessando atributos de uma estrutura

Para acessar um atributo de uma estrutura, usa-se o operador ponto (.), se a mesma for estática. E o operador seta (->) se a mesma for dinâmica.

Protótipo para objeto estático:

```
nome_estrutura nome_objeto;
nome_objeto.atributo;
```

Protótipo para objeto dinâmico:

```
nome_estrutura* ptr;
ptr = new nome_estrutura;
ptr ->atributo;
delete ptr;
```

O operador seta (->) é utilizado para acessar os elementos de um objeto dinâmico.

O exemplo abaixo ilustra a declaração e o uso de estruturas estáticas simples.

Listing G.1: Uso de struct.

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

/*
Uma struct é um conjunto de variáveis ou objetos reunidos.
Os objetos que compõem a struct podem ser de diferentes tipos.
No exemplo abaixo usa duas strings,
mas poderia usar string para
nome e int para matrícula.
*/
struct SPessoa
{
    string nome;
    string matricula;
};

int main ()
{
    string linha="-----\n";
    cout<<"Entre com o número de alunos da disciplina (ex=3):";
    int numeroAlunos;
    cin >> numeroAlunos;
    cin.get();

    //Cria um objeto professor do tipo SPessoa
    SPessoa professor;

    //Cria um vetor de objetos alunos do tipo SPessoa
    SPessoa aluno[numeroAlunos];

    cout << "Entre com o nome do professor:";
```

```

getline(cin, professor.nome);

cout << "Entre com a matricula do professor: ";
getline(cin, professor.matricula);

for(int contador = 0; contador < numeroAlunos; contador++)
{
    cout << "Aluno " << contador << endl;
    cout << "Entre com o nome do aluno: ";
    getline(cin, aluno[contador].nome);

    cout << "Entre com a matricula do aluno: ";
    getline(cin, aluno[contador].matricula);
}

cout << linha;
cout << "RELAÇÃO DE PROFESSORES E ALUNOS:" << endl;
cout << linha;

cout << "Nome do professor: " << professor.nome << "\n";
cout << "Matricula: " << professor.matricula << "\n";

for(int contador = 0; contador < numeroAlunos; contador++)
{
    cout << linha;
    cout << "Aluno " << contador << endl;
    cout << "Nome do aluno: " << aluno[contador].nome << endl;
    cout << "Matricula: " << aluno[contador].matricula << endl;
}

cin.get();
return 0;
}

/*
Novidades:
-Declaração e uso de estruturas estáticas
*/

/*
Saída:
-----
[andre@mercurio Cap2-Sintaxe]$ ./a.out
Entre com o número de alunos da disciplina (ex =3): 3
Entre com o nome do professor: P.C.Philippi
Entre com a matricula do professor: 1
Aluno 0
Entre com o nome do aluno: Fabio Magnani
Entre com a matricula do aluno: 2
Aluno 1
Entre com o nome do aluno: Liang Zhirong
Entre com a matricula do aluno: 3
Aluno 2
Entre com o nome do aluno: Savio
Entre com a matricula do aluno: 4
-----
RELAÇÃO DE PROFESSORES E ALUNOS:

```

```
-----
Nome do professor: P. C. Philippi
Matricula : 1
-----
```

```
Aluno 0
Nome do aluno: Fabio Magnani
Matricula : 2
-----
```

```
Aluno 1
Nome do aluno: Liang Zhirong
Matricula : 3
-----
```

```
Aluno 2
Nome do aluno: Savio
Matricula : 4
*/
```

G.1.4 Estruturas e funções²

Pode-se passar uma estrutura por cópia ou por referência para uma função.

Exemplo:

```
tipo função( nome_estrutura obj1); //por cópia
tipo função( nome_estrutura & obj2); //por referência
tipo função( nome_estrutura * obj2); //por ponteiro
```

G.1.5 Lista encadeada²

Consiste em colocar um ponteiro dentro de uma estrutura que aponta para a próxima estrutura.

Exemplo:

```
//Definição da estrutura
struct Carro
{
string nome(30);
float preco;
Carro *Ptr; //ponteiro para estrutura carro
};
//Cria objeto do tipo Carro apontado por ptr
Carro* ptr = new Carro;
//modifica valor da variável preco
ptr->preco = 10000.00;
...
//destrõe o objeto e zera o ponteiro
delete ptr; ptr=NULL;
```

G.1.6 Estruturas aninhadas²

Quando temos uma estrutura dentro de outra.

```
Exemplo:
//para estruturas aninhadas
struct Pessoa
{
int idade;
struct Familia
{
string nome_familia(255);
int numeroIrmaos;
}
}
void main()
{
//Cria objeto do tipo Pessoa
Pessoa Joao;
Joao.idade=21;
Joao.Familia.numeroIrmaos = 3;
Joao.Familia.nome_familia= "da Silva";
```

G.1.7 Sentenças para estruturas

- Uniões e estruturas dentro de classes são sempre public.
- Inicialmente não é permitida a inclusão de funções dentro da estrutura, hoje, uma struct aceita funções. Se quiser colocar funções dentro da estrutura use classes.

G.2 Uniões (union)

Uma union permite a um conjunto de objetos ocupar o mesmo local na memória, desta forma somente um objeto pode ser utilizado por vez. O espaço ocupado por uma union é correspondente ao do maior objeto definido.

Uma união pode ser considerada uma estrutura onde todos os objetos tem deslocamento zero, e cujo tamanho é suficiente para conter o seu maior membro.

Uniões não suportam herança, mas podem ter construtores.

Depois de definida uma union, o seu nome passa a ser um tipo do usuário.

Protótipo:

```
union nome_uniao
{
tipo1 v1;
tipo2 v2;
}
//Para criar um objeto da união faça:
nome_uniao obj;
//Para armazenar valores na união:
tipo1 x;
```

```
obj.v1 = x;
//Para acessar os atributos da união faça:
tipo1 y = obj.v1;
```

Exemplo:

```
union data
{
  int x;
  double y;
}
data x = 5;
data y = 5.0 + x ;
```

Observe que você pode colocar dois objetos definidos pelo usuário dentro de uma union, mas só pode usar um de cada vez. Veja a seguir um exemplo.

Listing G.2: Uso de union.

```
#include <iostream>
int main()
{
  union Nome_union
  {
    int raioHidraulico;
    double condutancia;
  };

  Nome_union obj;
  obj.raioHidraulico = 3;
  cout<<"raioHidraulico_□=□" << obj.raioHidraulico <<endl;
  cout<<"condutancia_□=□" << obj.condutancia <<endl;

  obj.condutancia = 5.0;
  cout<<"raioHidraulico_□=□" << obj.raioHidraulico <<endl;
  cout<<"condutancia_□=□" << obj.condutancia <<endl;

}
/*
Saída:
-----
[andre@mercurio Cap2-Sintaxe]$ ./a.out
raioHidraulico = 3
condutancia = 4.87444e-270
raioHidraulico = 0
condutancia = 5
*/
```

Você pode criar union's sem um nome, deste modo as variáveis definidas dentro da union podem ser acessadas diretamente, sem o uso do ponto(.), a esta união damos o nome de união anônima. Obviamente, como os nomes definidos dentro desta união podem ser utilizados diretamente no escopo em que foram declaradas, variáveis com o mesmo nome causarão ambiguidade.

Em alguns casos algum objeto pode ter o mesmo objetivo geral, mas seria interessante poder ter 2 nomes. veja o exemplo.

Listing G.3: Uso de union para apelidar atributo.

```
#include <iostream>
int main()
{
    union Nome_union
    {
        double raioHidraulico;
        double condutancia;
    };

    Nome_union obj;
    obj.raioHidraulico = 3.0;
    cout<<"raioHidraulico_=" << obj.raioHidraulico <<endl;
    cout<<"condutancia_=" << obj.condutancia <<endl;

    obj.condutancia = 5.0;
    cout<<"raioHidraulico_=" << obj.raioHidraulico <<endl;
    cout<<"condutancia_=" << obj.condutancia <<endl;
}

/*
Saída:
-----
[andre@mercurio Cap2-Sintaxe]$ ./a.out
raioHidraulico = 3
condutancia = 3
raioHidraulico = 5
condutancia = 5
*/
```

G.3 Enumerações (enumerated)

Uma enumeração é uma seqüência de objetos, do tipo int, que tem como objetivo enumerar algum processo. Uma enumeração definida pelo usuário é um tipo do usuário.

Exemplo:

```
//Declaração da enumeração
enum Dia{dom, seg, ter, qua, qui, sex, sab} ;
*/Onde:
Dia = tipo da enumeração,
dom=1, seg=2, ter=3, qua=4, qui=5, sex=6, sab=7
*/
//Criação de um objeto do tipo Dia com nome d
Dia d;
d = dom; //d=1
d = sab; //d=6
```

Você pode criar uma enumeração sem nome, neste caso os atributos internos podem ser acessados diretamente.

Exemplo:

```
enum {jan,fev,mar,abr,mai,jun,jul,ago,set,out,nov,dez};  
mes = jan; //se refere ao mês de janeiro
```

Uma enumeração permite sobrecarga de operador.

```
//Abaixo sobrecarga do operador++  
Dia& operator++(Dia& d)  
{  
    if(d == sab)  
        d = dom;  
    else  
        d++;  
}
```


Apêndice H

Bibliotecas de C

Apresenta-se neste capítulo exemplos de uso das bibliotecas de C.

H.1 <cmath>

Apresenta-se a seguir um exemplo de uso da biblioteca cmath de C.

Listing H.1: Uso de funções matemáticas.

```
//Manipulação de stream (io=input output stream)
#include <iostream>
using std::cout;
using std::endl;
using std::ios;

//Formatação de stream (manip = manipulador)
#include <iomanip>
using std::setiosflags;
using std::fixed;
using std::setprecision;

//Funções matemáticas
#include <cmath>

int main()
{
    cout
    << setiosflags( ios::fixed | ios::showpoint )<< setprecision( 1 )
    << "Raiz□quadrada" <<endl
    << "sqrt(" << 700.0 << ")□=□" << sqrt( 700.0 ) << endl
    << "sqrt(" << 7.0 << ")□=□" << sqrt( 7.0 ) << endl
    << "Exponencial□neperiana□(e^x)"<<endl
    << "exp(" << 1.0 << ")□=□" <<setprecision(6)<< exp(1.0)<< endl
    << "exp(" << setprecision( 1 ) << 2.0 << ")□=□"
    << setprecision( 6 ) << exp( 2.0 )<< endl
    << "logaritimo□neperiano"<<endl
    << "log(" << 2.718282 << ")□=□" <<setprecision(1)<<log(2.718282)<<endl
    << "log(" << setprecision( 6 )<< 7.389056 << ")□=□"
    << setprecision( 1 )<< log( 7.389056 ) << endl;
```

```

cout
<< "Logarítmo na base 10"<<endl
<< "log10(" << 1.0 << ")_=_ " << log10( 1.0 )<< endl
<< "log10(" << 10.0 << ")_=_ " << log10( 10.0 )<< endl
<< "log10(" << 100.0 << ")_=_ " << log10( 100.0 )<< endl
<< "Função valor absoluto, módulo"<<endl
<< "fabs(" << 13.5 << ")_=_ " << fabs( 13.5 )<< endl
<< "fabs(" << 0.0 << ")_=_ " << fabs( 0.0 )<< endl
<< "fabs(" << -13.5 << ")_=_ " << fabs( -13.5 ) << endl;

cout
<<"Truncamento"<<endl
<< "ceil(" << 9.2 << ")_=_ " << ceil( 9.2 )<< endl
<< "ceil(" << -9.8 << ")_=_ " << ceil( -9.8 )<< endl

<<"Arredondamento"<<endl
<< "floor(" << 9.2 << ")_=_ " << floor( 9.2 )<< endl
<< "floor(" << -9.8 << ")_=_ " << floor( -9.8 ) << endl;

cout
<<"Potenciação x^y"<<endl
<< "pow(" << 2.0 << ",_=" << 7.0 << ")_=_ "<< pow( 2.0, 7.0 )<< endl
<< "pow(" << 9.0 << ",_=" << 0.5 << ")_=_ " << pow( 9.0, 0.5 )<< endl
<< setprecision(3)
<< "fmod("<< 13.675 << ",_=" << 2.333 << ")_=_ "
<< fmod( 13.675, 2.333 ) << setprecision( 1 )<< endl

<<"Funções trigonométricas, sin, cos, tan"<<end
<< "sin(" << 0.0 << ")_=_ " << sin( 0.0 )<< endl
<< "cos(" << 0.0 << ")_=_ " << cos( 0.0 )<< endl
<< "tan(" << 0.0 << ")_=_ " << tan( 0.0 ) << endl;
    return 0;
}
/*
Novidade:
-----
-Uso de funções da biblioteca <cmath>
-Uso de funções de formatação de saída para tela
setprecision
*/
.

```

Apêndice I

Portabilidade

Veja a seguir um conjunto de sentenças para aumentar a portabilidade de seus programas.

- Vimos no capítulo 52 o uso dos compiladores e sistemas da GNU. A melhor maneira de desenvolver programas portáveis é usar a plataforma GNU.
- O arredondamento de número flutuantes pode variar de máquina para máquina.
- O caracter usado para encerrar uma entrada de dados é diferente nas plataformas DOS/Windows e Unix/Linux.
No Linux/Unix/Mac usa-se ctrl+d
No Windows usa-se ctrl+c.
- Use EOF para verificar se esta no final do arquivo.
- Numa plataforma 16 bits um int vai de -32767 a +32767.
Numa plataforma 32 bits um int vai de -2.147.483.648 a +2.147.483.648.
- Use a biblioteca STL em vez de bibliotecas de terceiros.
- Se você tem um ponteiro apontando para um objeto que foi deletado ou para um índice inválido em um vetor, pode ou não ocorrer um bug. Vai depender da sequência do programa, e da plataforma.
- Para saber o tamanho de um objeto use sizeof.
- O tamanho dos objetos padrões de C++ estão definidos em <limits>.
- O sizeof de um objeto pode ser maior que a soma do sizeof de seus atributos. Isto ocorre em função do alinhamento dos atributos. O alinhamento varia de máquina para máquina.
- No Linux/GNU já existe int de 64 bits
long long obj_int_64bits;
- Use a stl.
- Se você quer um int no intervalo >32000, use long int.
- Se quiser um int <32000 use short.

- Use 0 para representar false e 1 para representar true.
- Use `if(ch == 'A')` e não `if(ch == 65)`.
- O `sizeof` de uma classe pode variar entre diferentes plataformas.
- Evite usar `reinterpret_cast`.

Apêndice J

Bug / Debug

Apresenta-se neste apêndice o conceito de bug, o uso de assert e dicas para evitar bugs em seus programas.

J.1 O que é um bug?

Um dos primeiros computadores montados pelo homem, tinha uma quantidade enorme de fios. Estes computadores começaram a apresentar falhas. Depois de muitas análises, os engenheiros e programadores descobriram que as falhas estavam ocorrendo porque alguns insetos (BUGS) comeram alguns fios.

Assim, um bug é uma falha em um programa.

O que é debugar um programa?

Debugar um programa é eliminar os erros do programa.

J.2 Uso de assert

A macro assert é definida em <cassert> e é usada para testar uma expressão. Se a expressão é falsa (0) imprime uma mensagem de erro informando a linha do programa e a seguir chama abort().

Exemplo:

```
int* x = NULL;
x = new int [50];
assert( x != NULL);
```

Se x for NULL encerra o programa, se x não for NULL continua.

Para que as instruções assert sejam consideradas, deve-se definir a variável DEBUG. Para desabilitar (desconsiderar as instruções assert) defina a variável NDEBUG.

Exemplo:

```
#define DEBUG
#define NDEBUG
```

DICA: Só use assert no período de debugagem. Para testar alocação de ponteiro use instruções separadas.

J.3 Sentenças para evitar bugs

Apresenta-se a seguir uma lista de dicas para reduzir o número de bugs em seu programa. Boa parte destas dicas foram extraídas da fonte, [Maguire, 1994].

- De um modo geral, um programador pode encontrar 90% de seus bugs, uma equipe de programação 96% , com o uso de versões beta, pode-se encontrar 99.9% dos bugs.
- Para criar um comando inútil use NULL
Exemplo:
`while (x == y) NULL;`
- Cuidado com `if(ch = '\t')`
o correto é `if(ch == '\t')`.
- Se a variável não muda, use `const`.
- Trocar `if(x)` por `if(x!= NULL)`.
- Sempre crie protótipos para as funções.
- Nunca use `goto`.
- Ative todas as mensagens de warning e erro do compilador.
- Verifique se os parâmetros são passados na ordem correta.
- Verifique os intervalos dos parâmetros.
- Verifique o retorno das funções.
- 063 é o octal 51 e não 63.
- Use o conteúdo de `limits.h`.
- Para reduzir o número de testes, verificar as entradas (limites inferior/superior).

Entradas do usuário (teclado, mouse)

Acesso a disco (rw?)

Alocação desalocação de memória.

- Use a variável `DEBUG`
`# ifdef DEBUG`
`testes`
`# endif`
- Gere duas versões de seu programa, uma de depuração e uma para usuário final.
- Use instruções `assert`
`assert(p1 != NULL & & p2 !=NULL);`

- Documentar o porque da instrução `assert`.
- Não use `assert` para testar alocação de memória, pois sem o `#` define `DEBUG` o teste de alocação não entra no programa final.
- Compare o resultado de suas rotinas com outras que fazem a mesma coisa.
- Verifique as suposições assumidas.
- Após delatar um objeto faça seu ponteiro apontar para `NULL`.
- Verifique erros de `overflow/underflow`.
- Verifique conversões de dados.
- Verifique a precedência dos operadores.
- Use parenteses extras para deixar o código mais claro.
- `EOF` é um valor negativo.
- Verifique as conversões automáticas (`char->int`).
- Evite usar `unsigned int`
`unsigned int z= x-y; //x=3 y=6 z=-3? z=?`
- Se uma função deve testar uma condição, faça uma única vez e pense em dividir a função em 2 blocos.
- Escreva códigos para programadores medianos (evite códigos rebuscados).
- Erros não desaparecem do nada.
- Não deixa para arrumar o erro depois, arrume já.
- Mantenha um arquivo de `change.log` onde serão anotadas as alterações realizadas.
- Mantenha um arquivo de bugs, onde são relatados os bugs, se os mesmos foram encontrados e as soluções adotadas.
- Após escrever uma função (ou objeto) teste o mesmo.
- Se um atributo `z` é `private` você sabe que ele só pode ser utilizado na classe onde foi declarada. Assim, se ocorrer um bug com o atributo `z`, você só precisa conferir o código da classe. Se um atributo é protegido, ele só pode ser utilizado na classe onde foi declarado e pelas herdeiras, assim, ao procurar um bug, você só precisa procurar o bug na classe onde o atributo foi declarado e nas classes herdeiras. Se um atributo público esta causando um bug, você terá de procurar o mesmo em toda a hierarquia de classes e no restante do seu programa. Observe que quando mais encapsulado o seu código, maior a facilidade para encontrar erros e bugs.

- A noção de vetor em C é de nível muito baixo, o que faz com que o mesmo seja extremamente versátil, mas que exige do programador certos cuidados. Um vetor tem tamanho fixo determinado em tempo de compilação, é unidimensional (bi e tri-dimensional se refere a vetores de vetores); um vetor não é autodescrito, ou seja, um ponteiro para um vetor não indica o tamanho do vetor.
- Se existe acesso a disco o usuário deve ter permissão de escrita e leitura.
- E se o HD estiver cheio?.
- Use diferentes compiladores.
- Use bibliotecas de terceiros e que sejam muito conhecidas e testadas.
- Todo switch deve ter uma opção default.
- Sempre inicializar os ponteiros com NULL.
- Defina os atributos como private, só mude para protected e public se realmente necessário.
- Não confundir `char = 'c'` com `char = "mensagem"`.

Apêndice K

Glossário

Abstração: Processo de criar uma superclasse pela extração de qualidades comuns ou características gerais de uma ou mais classes ou objetos específicos.

Ação: (em modelagem dinâmica) operação instantânea, associada a um evento.

Agente: veja objeto ativo.

Agregação: forma especial de associação, entre o todo e suas partes, no qual o todo é composto pelas partes.

Agregado fixo: quando o número de partes é definido.

Agregado recursivo: quando o objeto pode conter a si próprio.

Agregado variável: quando o número de partes é indefinido.

Algoritmo: conjunto de etapas ordenadas de forma específica, usadas para solucionar um problema, tal como uma fórmula matemática ou uma série de instruções de um programa.

Ambiente Windows: Computador que roda sobre um sistema operacional com janelas múltiplas na tela.

Amigo: classes amigas, permitem o compartilhamento de métodos e atributos.

Análise orientada a objeto: Análise dos requisitos de um sistema em termos de objetos reais. Realizada sem considerar os requisitos de implementação.

Ancestral imediato: aquele que é identificado na declaração do objeto.

Arquitetura: estrutura geral de um sistema, incluindo a sua divisão em subsistemas e suas alocações para tarefas e processadores.

Assinatura: para um atributo o tipo do atributo, para uma operação o nome, parâmetros, e o retorno.

Associação derivada: definida em termos de outras associações.

Associação qualificada: associação que relaciona duas classes e um qualificador.

Associação ternária: associação entre três classes.

Associação: relacionamento entre duas ou mais classes descrevendo um grupo de ligações com estruturas e semânticas comuns.

Atividade: (em modelagem dinâmica) operação que leva tempo para terminar. Estão relacionadas a estados e representam acontecimentos do mundo real.

Atributo de classe: atributo que existe na classe e não nos objetos. Todos os objetos da classe tem acesso ao atributo de classe, mas este é único.

Atributo de evento: dados transportados por um evento.

Atributo de ligação: Atributo que existe em função de uma ligação entre dois objetos.

Atributo derivado: Atributo que é calculado a partir de outros atributos.

Atributo: uma propriedade ou característica de um objeto.

Banco de dados: uma coleção de dados armazenados e gerenciados eletronicamente.

Biblioteca de classes: Uma coleção de classes genéricas que podem ser adaptadas e determinadas para uma aplicação particular.

Browser: ferramenta que acompanha a linguagem permitindo ao programador visualizar a hierarquia e editar o código em linguagens orientadas ao objeto.

Caixa Preta: metáfora usada em engenharia que descreve um dispositivo onde os componentes internos são desconhecidos pelo usuário. Os objetos são como caixas pretas em que seus funcionamentos internos ficam ocultos aos usuários e programadores.

Cancelar: definir um método em uma classe derivada que substitui o método ancestral.

Característica: é uma palavra genérica para atributos e métodos.

Cenário: (em modelagem dinâmica) seqüência de eventos que ocorre durante uma determinada execução do sistema.

Classe abstrata: classe de não pode gerar objetos.

Classe concreta: classe que pode gerar objetos.

Classe derivada: uma subclasse em C++.

Classe filho: veja subclasse.

Classe Pai: veja superclasse.

Classes: sinônimo de tipos de objeto (Fábrica de objetos).

Coerência: propriedade de uma entidade, como uma classe, uma operação ou módulo, tal que ela se organize em um plano consistente e todas as suas partes se ajustem no sentido de um objetivo comum.

Coleção de lixo: rotina de gerenciamento de memória que pesquisa a memória por segmentos de programa, dados ou objetos que não se encontram mais ativos e recupera o espaço não utilizado.

Compilador: programa que traduz uma linguagem de programação para a linguagem de máquina.

Concorrência: a capacidade de programas orientados a objeto de se comunicarem em multitarefa.

Concorrente: duas ou mais tarefas, atividades ou eventos cujas execuções podem se sobrepor no tempo.

Condição: (em modelagem dinâmica) função booleana de valores de objetos válidos durante um intervalo de tempo.

Conseqüência da implementação da hereditariedade e do polimorfismo. Quando um programa orientado ao objeto é executado, as mensagens são recebidas por objetos. O método para manipulação de uma mensagem é armazenado no alto de uma biblioteca de classes. O método é localizado dinamicamente quando necessário e a ligação ocorre no último momento possível.

Construtor: método especial que inicializa todos os atributos de um objeto.

Consulta, é uma operação que quando executada não altera o objeto.

Dados persistentes: dados que continuam existindo mesmo após encerrada a execução de um programa.

Debugger: programa que permite ao usuário corrigir erros de software por intermédio do exame e alteração do conteúdo da memória, e iniciar ou parar a execução em um local predeterminado ou breakpoint.

Delegação: mecanismo de implementação no qual um objeto, em resposta a uma operação nele próprio, repassa a operação para outro objeto.

Depurador: Serve para localizar e alterar erros de lógica de um programa ou para correções em um programa solicitando um símbolo determinado do código fonte.

Designe orientado a objeto: tradução da estrutura lógica de um sistema em uma estrutura física composta por objetos de software.

Destrutor: libera os objetos dinamicamente alocados e destroi o objeto. Podem ser explícitos ou implícitos, se explícitos (declarados) podem liberar outros tipos de dados e realizar outras funções.

Diagrama de eventos: diagrama que mostra o remetente e o receptor.

Diagrama de fluxo de dados: diagrama que mostra o movimento de dados e seu processamento manual ou por computador.

Diagrama de fluxo de dados: representação gráfica do modelo funcional, mostrando as dependências entre os valores e o cálculo de valores de saída a partir de valores de entrada sem preocupações com, quando e se as funções serão executadas.

Diagrama de objetos: diagrama que mostra o relacionamento de objetos entre si.

Diagrama de objetos: representação gráfica de objetos mostrando os relacionamentos, atributos e operações.

Dicionário de dados: parágrafo que descreve uma classe, seus atributo, operações e ligações.

Discordância de nome: conflito que pode ocorrer em hereditariedade múltipla quando o mesmo método ou atributo de instância for herdado de classes múltiplas.

Encapsulamento: casamento do código com os dados dentro de uma unidade de objeto. Isto representa a modularidade aplicada aos dados. O encapsulamento torna invisíveis os dados para o usuário, embora os métodos permaneçam visíveis.

Escala: O relacionamento entre 3 elementos é chamado escala.

Especialização: a criação de subclasses a partir de uma superclasse através do refinamento da superclasse.

Estado: valores dos atributos e ligações de um objeto em um determinado momento.

Evento: (em modelagem dinâmica) algo que acontece instantaneamente em um momento de tempo.

Expansibilidade: expandir POO sem ter em mãos o código-fonte.

Extensão: (em generalização) o acréscimo de novas características através de uma subclasse.

Extensibilidade: habilidade de um programa ou sistema em ser facilmente alterado para que possa tratar novas classes de entrada.

Folha: mecanismo de subdivisão de um modelo de objetos em uma série de páginas.

Framework: uma biblioteca de classe "afinada" especialmente para uma determinada categoria de aplicação.

Função virtual: método especial chamado por intermédio de uma referência de classe básica ou indicador, carregado dinamicamente em tempo de execução.

Gerenciamento de memória: a maneira pela qual o computador trata a sua memória. Inclui proteção de memória e quaisquer outras técnicas de memória virtual.

Gráficos orientados a objeto: programas que tem a função de desenhar e que são apresentados ao usuário sob a forma de objetos na tela.

Handle: variável usada pelo sistema operacional para identificar um objeto.

Herança múltipla: permite a um objeto ter mais de um pai, herdando métodos e atributos de mais de um pai.

-
- Herança repetida:** Quando um objeto é descendente de outro por mais de uma caminho.
- Herança:** A propriedade de todos os tipos de objetos que permite a um tipo ser definido como herdando todas as definições e atributos e métodos contidos no pai.
- Hereditariedade:** mecanismo usado para compartilhar métodos e tipos de dados automaticamente entre classes, subclasses, e objetos. Não é encontrado em sistemas de procederes. Permite programar apenas as diferenças de classes previamente definidas.
- Hierarquia de objetos (estrutura):** é um diagrama que mostra o relacionamento dos objetos.
- Hierarquia:** Descrição de um sistema que conta com uma estrutura construída em níveis diferentes. Os níveis mais altos controlam os mais baixos.
- Ícone:** representação gráfica de um objeto.
- Identidade do objeto:** algo sobre o objeto que permanece invariável entre todas as possibilidades de modificação de seu estado. Pode ser usado para indicar um objeto.
- Informação escondida/oculta:** recurso de programação pelo qual a informação dentro de um módulo permanece privada a ele. Estratégia de design que visa maximizar a modularidade ocultando o maior número possível de informações dentro dos componentes de um design.
- Instância:** objeto que faz parte de uma classe.
- Instanciação:** processo de criação de instâncias a partir de classes.
- Invariante:** declaração sobre alguma condição ou relacionamento que deve ser sempre verdadeiro.
- Kernel:** parte fundamental de um programa como um sistema operacional, residente todo o tempo em memória.
- Ligação a posteriori (dinâmica):** Um método que permite chamar as procederes cujo endereço não é conhecido em tempo de compilação /linkedição. O endereço só é conhecido durante a execução.
- Ligação a priori/ anterior:** método tradicional de compilação pelo qual os endereços das procederes e funções são determinados em tempo de linkedição/compilação.
- Ligação estática:** veja ligação a priori.
- Ligação:** processo de organizar um programa para solucionar todas a conexões entre seus componentes. Estaticamente, a ligação obtém essas conexões antes de rodar o programa. Dinamicamente a ligação ocorre quando o programa esta rodando.
- Linguagem concorrente:** linguagem que permite a execução simultânea de objetos múltiplos, geralmente com arquitetura de hardware paralela.
- Linguagem não procedural:** linguagem de programação que gera a lógica para o programa diretamente a partir da descrição do problema pelo usuário, em vez de um conjunto de procederes baseadas em lógica de programação tradicional.

Linguagem orientada a objeto: linguagem de computador que suporta objetos, classes, métodos, mensagens e hereditariedade. As características secundárias podem incluir hereditariedade múltipla, ligação dinâmica e polimorfismo.

Linguagem procedural: linguagem de programação como COBOL, FORTRAN, BASIC, C e Pascal, baseada na utilização de ordem particular de ações e que tem conhecimento das operações de processamento de dados e técnicas de programação.

Manutenção: capacidade de um programa ou sistema para transformar reparos de bugs e aumentar a funcionalidade a fim de satisfazer as necessidades do usuário.

Mensagem: solicitação enviada a um objeto para alterar seu estado ou retornar um valor. A mesma mensagem pode ser enviada para objetos diferentes porque eles simplesmente informam a um objeto o que fazer. Os métodos, definidos dentro de um objeto receptor, determinam como será executada a solicitação. Veja método/polimorfismo.

Metaclasse: classe que descreve outras classes.

Metadados: são dados que descrevem outros dados. Por exemplo uma definição de uma classe.

Método de implementação: (estilo) método que realiza o processamento sem realizar qualquer tomada de decisões.

Método estático: implementado usando a ligação a priori.

Método político: método que realiza o processamento de tomadas de decisões. Não realiza processamento de dados.

Método Virtual: implementado usando a ligação a posteriori.

Método: uma procedure ou função que é definida como pertencendo a um tipo de objeto. Implementa a resposta quando uma mensagem é enviada a um objeto. Os métodos determinam como um objeto responderá a uma mensagem. Veja mensagens.

Metodologia: em engenharia de software, é o processo de produção organizada de um programa, utilizando técnicas e convenções conhecidas.

Modelo funcional: descrição dos aspectos de um sistema que transforma valores utilizando funções, mapeamentos, restrições e dependências funcionais.

Modularidade: a construção de um programa em módulos.

Multiplicidade: número de instâncias de uma classe que podem se relacionar a uma única instância de outra classe.

Objeto ativo: um objeto que monitora eventos ocorrendo em uma aplicação e assume a ação por si próprio. Às vezes chamado de agente.

Objeto passivo: objeto que atua somente por solicitação.

Objeto polimórfico: o descendente herda a forma do ancestral (métodos), mas pode redefinir os métodos assumindo outras formas (polimorfismo).

Objeto: elemento primário em um programa orientado a objeto. Objetos são entidades que encapsulam dentro de si próprios os dados que descrevem o objeto e as instruções para operar estes dados.

Operação abstrata: operação declarada mas não implementada.

Operação de classe: operação que é realizada pela classe e não pelas instâncias desta. Só pode manipular os atributos de classe.

Operação: função ou transformação que pode ser aplicada por um objeto.

Orientado por Objeto: termo para prática de programação ou compiladores que agrupam elementos individuais de programação em hierarquias de classes, permitindo que objetos do programa compartilhem o acesso a dados e procedimentos sem redefinição.

Pai-filho: maneira de expressar a relação entre classes e subclasses. As classes filho ou subclasses herdam os métodos e atributos de instância da classe pai, por meio da hereditariedade múltipla um filho pode ter diversos pais.

Papel: uma extremidade de uma associação.

Paradigmas Híbridos: linguagens híbridas como o C++, que fazem uso de técnicas de programação OOP e estruturada.

Pilha: árvore binária completa em que os nodos contém chaves de pesquisa organizadas em ordem descendente.

Polimorfismo: propriedade de compartilhar uma ação simples. Cada classe derivada implementa os métodos de forma apropriada as suas necessidades. Habilidade de a mesma mensagem ser interpretada de maneiras diferentes quando recebida por objetos diferentes. A impressão de uma mensagem, por exemplo, quando enviada a uma figura ou diagrama, aciona um método ou implementação diferente daquele que a mesma mensagem de impressão envia a um documento.

Pré-processador: programa de software que executa procedimentos preliminares na entrada de dados antes da execução do programa principal. Por exemplo, o código fonte C++ é comumente pré-processado e traduzido para o código fonte C antes de ser compilado.

Privado: os métodos e dados pertencentes a uma classe são privados por default. Os elementos privados só podem ser acessados por métodos que pertençam a classe, desde que esta não seja descendente.(só acessado pela classe).

Problema:,

Processamento distribuído: sistema de computadores ligados por uma rede de comunicações com cada sistema gerenciando sua própria carga local e a rede suportando o sistema como um todo.

Processo: alguma coisa que transforma valores de dados.

Programa: conjunto de instruções que informam ao computador o que fazer. Um programa é escrito em uma linguagem de programação e é convertido em linguagem de máquina por meio de softwares chamados montadores e compiladores.

Programação estruturada: filosofia de programação voltada ao gerenciamento de complexidade por meio da formalização e padronização da metodologia de programação. A programação estruturada é caracterizada pela sua apresentação top-down:

Programação orientada ao objeto: metodologia usada para a criação de programas por intermédio da utilização de objetos auto suficientes com dados e o comportamento encapsulados e que atuam por meio de solicitação e interação com outros enviando e devolvendo mensagens.

Programação top-down: metodologia que cria um programa modular de estrutura hierárquica. Primeiro, o desenhista projeta, codifica e testa um módulo que representa a estrutura do programa e depois continua da mesma maneira criando módulos de nível mais baixo, que representa suas subfunções.

Programação visual: categoria genérica de aplicativos que executam programação gráfica e seus efeitos visuais ao usuário. Em pacotes de desenho, por exemplo, os objetos podem ser desenhados, aumentados e até modificados por meio de manipulações diretas da imagem na tela e não pela alteração de dados numéricos em uma tabela de dimensões.

Protegido: podem ser acessados pela classe e pelas classes derivadas.

Protocolo: conjunto de mensagens as quais o objeto pode responder.

Público: os métodos declaradas como public podem ser acessadas pela classe, classe derivadas, e por aplicações (dentro de main() por exemplo).

Qualificador: atributo de um objeto que faz a distinção entre o conjunto de objetos na extremidade muitos de uma associação.

Recursão: habilidade de uma sub-rotina ou módulo de programa em chamar a si mesma.

Redefinição: propriedade das classes derivadas de modificar os métodos da classe pai. Assim duas classes irmãs (filhas do mesmo pai) podem ter métodos com o mesmo nome mas com realizações diferentes. Representa um conceito que faz sentido no domínio da aplicação.

Restrição: (em generalização) limitação que uma sub-classe coloca no valor de um atributo contido em uma superclasse.

Robusto: um soft robusto não é facilmente destruído por erros em seus pressupostos. Conta com verificações internas que objetivam eliminar bugs.

Simulação: representação matemática da interação de objetos do mundo real.

Sobrecarga de operador:

SubClasse: refinamento de uma classe em uma outra mais especializada. Às vezes refere-se à classe derivada ou filho. Os métodos comuns e os tipos de dados são armazenados por intermédio da retirada de uma classe o máximo possível para que possam ser herdados por todas as classes relevantes.

SubRotina: veja procedure.

Superclasse: em uma hierarquia de hereditariedade, uma classe mais genérica que armazena atributos e métodos que podem ser herdados por outras classes. Algumas vezes tratada como classe base ou pai.

Tabelas dos métodos virtuais (VMT): tabela que aparece no segmento de dados de cada tipo de objeto virtual. A VMT contém o tamanho do tipo do objeto (tamanho do registro) e os ponteiros para as procedures e funções que implementam os métodos do tipo de objeto.

this: identificador invisível automaticamente declarado dentro do objeto. Pode ser usado para resolver conflitos entre identificadores pela classificação dos campos de dados que pertencem a um objeto do método.

Tipificação: propriedade de uma linguagem de distinguir com clareza os tipos definidos. A tipificação forte (C++) ajuda a desenvolver soft's mais robustos.

Tipo ancestral: todo tipo herdado por qualquer outro tipo de objeto.

Tipo de dados abstratos: conjunto de estruturas de dados (tipos de dados) definido em termos de recursos de estruturas e de operações executadas sobre elas. Na POO, os tipos de objetos são tipos de dados abstratos.

Unix: sistema operacional multi-usuário e multitarefa que trata os dados que foram designados a ela até aparecer um novo valor ou o programa terminar sua execução.

Variável de instância: dado contido em um objeto que descreve propriedades do objeto que a possui.

Variável global: variável acessível a todos os módulos de um programa.

Variável: uma estrutura em memória que trata os dados que foram designados a ela até aparecer um novo valor ou o programa terminar a sua execução.

Virtual: ambiente simulado ou conceitual. Realidade virtual, por exemplo, é uma realizada simulada.

Windows: área de visualização separada em uma tela de exibição fornecida pelo software. Os sistemas operacionais podem mostrar janelas múltiplas na tela, permitindo que o usuário mantenha vários programas aplicativos ativados e visíveis ao mesmo tempo. Os programas aplicativos individuais também contam com janelas múltiplas, oferecendo capacidade de visualização para mais de um documento, planilha ou arquivo de dados.

Apêndice L

Links Para Sites em C++

Apresenta-se a seguir uma lista de links relacionados a programação em C++.

Bookmark

O bookmark que utilizo para acessar os sites de programação esta disponibilizado em:

www.lmpt/ufsc.br/~andre/ApostilaProgramacao/bookmarks-prog.html.

HOWTO

Apresenta-se a seguir uma lista de HOWTO's relacionados direta ou indiretamente com programação em C++, Linux. Estes HOWTO's podem ser obtidos no site:

<http://www.tldp.org/HOWTO/HOWTO-INDEX/howtos.html>.

- C++ Programming HOW TO
- C++ Beautiful HOWTO
- CVS RCS HOW TO (document for Linux Source Code Control System)
- GCC HOWTO
- Program Library HOWTO
- Kernel HOWTO (kernel do Linux)
- Beowulf HOWTO
- Parallel Processing HOWTO
- Serial Programming HOWTO
- Glibc 2 HOWTO
- Software Release Practice HOWTO
- Bash Prog Intro HOWTO, BASH Programming Introduction HOWTO
- C editing with VIM HOWTO
- Emacs Beginner HOWTO

Exemplos

- Você pode baixar um conjunto de exemplos de programas em C++ em:
- www.deitel.com
- <ftp://ftp.cs.rpi.edu/pub/stl>

Tutoriais

Uma lista completa de tutoriais pode ser encontrada em:

<http://www.mysteries-megasite.com/linux/tutorials.html>

- Tutorial de C++ para programadores de C: <http://www.4p8.com/eric.brasseur/cppcen.html>
- <http://www.cplusplus.com/doc/tutorial/>
- <http://www.intap.net/~drw/cpp/index.htm>
- <http://www.gtk.org/tutorial/>

Sintaxe de C++

- Common c++: <http://www.voxilla.org/projects/projape.html>
- C++ libs: <http://www.thefreecountry.com/developercity/freelib.html>
- C++ Tools: <http://development.freeservers.com>
- C++ Tools CUJ: <http://www.cuj.com/code>
- C++ libs Univ of vaasa: http://garbo.uwasa.fi/pc/c_lang.html

STL - Standart Template Library

- <http://www.roguewave.com/support/docs/stdug/index.cfm>

Programação para Linux

- Portability Guide:
http://www.angelfire.com/country/aldev0/cpphowto/cpp_PortabilityGuide.html

Grupo de Linux na UFSC

- <http://www.softwarelivre.ufsc.br>

Apêndice M

Licença Pública Geral GNU

Versão 2, junho de 1991.

This is an unofficial translation of the GNU General Public License into Brazilian Portuguese. It was not published by the Free Software Foundation, and does not legally state the distribution terms for software that uses the GNU GPL – only the original English text of the GNU GPL does that. However, we hope that this translation will help Brazilian Portuguese speakers understand the GNU GPL better.

Esta é uma tradução não-oficial da Licença Pública Geral GNU ("GPL GNU") para o português do Brasil. Ela não foi publicada pela Free Software Foundation, e legalmente não afirma os termos de distribuição de software que utiliza a GPL GNU – apenas o texto original da GPL GNU, em inglês, faz isso. Contudo, esperamos que esta tradução ajude aos que utilizam o português do Brasil a entender melhor a GPL GNU.

Copyright (C) 1989, 1991 Free Software Foundation, Inc. 675 Mass Ave, Cambridge, MA 02139, USA

A qualquer pessoa é permitido copiar e distribuir cópias desse documento de licença, desde que sem qualquer alteração.

M.1 Introdução

As licenças de muitos software são desenvolvidas para restringir sua liberdade de compartilhá-lo e mudá-lo. Contrária a isso, a Licença Pública Geral GNU pretende garantir sua liberdade de compartilhar e alterar software livres – garantindo que o software será livre e gratuito para os seus usuários. Esta Licença Pública Geral aplica-se à maioria dos software da Free Software Foundation e a qualquer outro programa cujo autor decida aplicá-la. (Alguns outros software da FSF são cobertos pela Licença Pública Geral de Bibliotecas, no entanto.) Você pode aplicá-la também aos seus programas.

Quando nos referimos a software livre, estamos nos referindo a liberdade e não a preço. Nossa Licença Pública Geral foi desenvolvida para garantir que você tenha a liberdade de distribuir cópias de software livre (e cobrar por isso, se quiser); que você receba o código-fonte ou tenha acesso a ele, se quiser; que você possa mudar o software ou utilizar partes dele em novos programas livres e gratuitos; e que você saiba que pode fazer tudo isso.

Para proteger seus direitos, precisamos fazer restrições que impeçam a qualquer um negar estes direitos ou solicitar que você deles abdique. Estas restrições traduzem-se em certas responsabilidades para você, se você for distribuir cópias do software ou modificá-lo.

Por exemplo, se você distribuir cópias de um programa, gratuitamente ou por alguma quantia, você tem que fornecer aos recebedores todos os direitos que você possui. Você tem que garantir que eles também recebam ou possam obter o código-fonte. E você tem que mostrar-lhes estes termos para que eles possam conhecer seus direitos.

Nós protegemos seus direitos em dois passos: (1) com copyright do software e (2) com a oferta desta licença, que lhe dá permissão legal para copiar, distribuir e/ou modificar o software.

Além disso, tanto para a proteção do autor quanto a nossa, gostaríamos de certificar-nos que todos entendam que não há qualquer garantia nestes software livres. Se o software é modificado por alguém mais e passado adiante, queremos que seus recebedores saibam que o que eles obtiveram não é original, de forma que qualquer problema introduzido por terceiros não interfira na reputação do autor original.

Finalmente, qualquer programa é ameaçado constantemente por patentes de software. Queremos evitar o perigo de que distribuidores de software livre obtenham patentes individuais, o que tem o efeito de tornar o programa proprietário. Para prevenir isso, deixamos claro que qualquer patente tem que ser licenciada para uso livre e gratuito por qualquer pessoa, ou então que nem necessite ser licenciada.

Os termos e condições precisas para cópia, distribuição e modificação se encontram abaixo:

M.2 Licença pública geral GNU termos e condições para cópia, distribuição e modificação

0. Esta licença se aplica a qualquer programa ou outro trabalho que contenha um aviso colocado pelo detentor dos direitos autorais informando que aquele pode ser distribuído sob as condições desta Licença Pública Geral. O "Programa" abaixo refere-se a qualquer programa ou trabalho, e "trabalho baseado no Programa" significa tanto o Programa em si como quaisquer trabalhos derivados, de acordo com a lei de direitos autorais: isto quer dizer um trabalho que contenha o Programa ou parte dele, tanto originalmente ou com modificações, e/ou tradução para outros idiomas. (Doravante o processo de tradução está incluído sem limites no termo "modificação".) Cada licenciado é mencionado como "você".

Atividades outras que a cópia, a distribuição e modificação não estão cobertas por esta Licença; elas estão fora de seu escopo. O ato de executar o Programa não é restringido e o resultado do Programa é coberto apenas se seu conteúdo contenha trabalhos baseados no Programa (independentemente de terem sido gerados pela execução do Programa). Se isso é verdadeiro depende do que o programa faz.

1. Você pode copiar e distribuir cópias fiéis do código-fonte do Programa da mesma forma que você o recebeu, usando qualquer meio, deste que você conspícua e apropriadamente publique em cada cópia um aviso de direitos autorais e uma declaração de inexistência de garantias; mantenha intactas todos os avisos que se referem a esta Licença e à ausência total de garantias; e forneça a outros recebedores do Programa uma cópia desta Licença, junto com o Programa.
2. Você pode modificar sua cópia ou cópias do Programa, ou qualquer parte dele, assim gerando um trabalho baseado no Programa, e copiar e distribuir essas modificações ou trabalhos sob os termos da seção 1 acima, desde que você também se enquadre em todas estas condições:

- a) Você tem que fazer com que os arquivos modificados levem avisos proeminentes afirmando que você alterou os arquivos, incluindo a data de qualquer alteração.
- b) Você tem que fazer com que quaisquer trabalhos que você distribua ou publique, e que integralmente ou em partes contenham ou sejam derivados do Programa ou de suas partes, sejam licenciados, integralmente e sem custo algum para quaisquer terceiros, sob os termos desta Licença.
- c) Se qualquer programa modificado normalmente lê comandos interativamente quando executados, você tem que fazer com que, quando iniciado tal uso interativo da forma mais simples, seja impresso ou mostrado um anúncio de que não há qualquer garantia (ou então que você fornece a garantia) e que os usuários podem redistribuir o programa sob estas condições, ainda informando os usuários como consultar uma cópia desta Licença. (Exceção: se o Programa em si é interativo mas normalmente não imprime estes tipos de anúncios, seu trabalho baseado no Programa não precisa imprimir um anúncio.)

Estas exigências aplicam-se ao trabalho modificado como um todo. Se seções identificáveis de tal trabalho não são derivadas do Programa, e podem ser razoavelmente consideradas trabalhos independentes e separados por si só, então esta Licença, e seus termos, não se aplicam a estas seções quando você distribui-las como trabalhos em separado. Mas quando você distribuir as mesmas seções como parte de um todo que é trabalho baseado no Programa, a distribuição como um todo tem que se enquadrar nos termos desta Licença, cujas permissões para outros licenciados se estendem ao todo, portanto também para cada e toda parte independente de quem a escreveu.

Desta forma, esta seção não tem a intenção de reclamar direitos ou contestar seus direitos sobre o trabalho escrito completamente por você; ao invés disso, a intenção é a de exercer o direito de controlar a distribuição de trabalhos, derivados ou coletivos, baseados no Programa.

Adicionalmente, a mera adição ao Programa de outro trabalho não baseado no Programa (ou de trabalho baseado no Programa) em um volume de armazenamento ou meio de distribuição não faz o outro trabalho parte do escopo desta Licença.

3. Você pode copiar e distribuir o Programa (ou trabalho baseado nele, conforme descrito na Seção 2) em código-objeto ou em forma executável sob os termos das Seções 1 e 2 acima, desde que você faça um dos seguintes:

- a) O acompanhe com o código-fonte completo e em forma acessível por máquinas, que tem que ser distribuído sob os termos das Seções 1 e 2 acima e em meio normalmente utilizado para o intercâmbio de software; ou,
- b) O acompanhe com uma oferta escrita, válida por pelo menos três anos, de fornecer a qualquer um, com um custo não superior ao custo de distribuição física do material, uma cópia do código-fonte completo e em forma acessível por máquinas, que tem que ser distribuído sob os termos das Seções 1 e 2 acima e em meio normalmente utilizado para o intercâmbio de software; ou,
- c) O acompanhe com a informação que você recebeu em relação à oferta de distribuição do código-fonte correspondente. (Esta alternativa é permitida somente

em distribuição não comerciais, e apenas se você recebeu o programa em forma de código-objeto ou executável, com oferta de acordo com a Subseção b acima.)

O código-fonte de um trabalho corresponde à forma de trabalho preferida para se fazer modificações. Para um trabalho em forma executável, o código-fonte completo significa todo o código-fonte de todos os módulos que ele contém, mais quaisquer arquivos de definição de "interface", mais os "scripts" utilizados para se controlar a compilação e a instalação do executável. Contudo, como exceção especial, o código-fonte distribuído não precisa incluir qualquer componente normalmente distribuído (tanto em forma original quanto binária) com os maiores componentes (o compilador, o "kernel" etc.) do sistema operacional sob o qual o executável funciona, a menos que o componente em si acompanhe o executável.

Se a distribuição do executável ou código-objeto é feita através da oferta de acesso a cópias de algum lugar, então ofertar o acesso equivalente a cópia, do mesmo lugar, do código-fonte equivale à distribuição do código-fonte, mesmo que terceiros não sejam compelidos a copiar o código-fonte com o código-objeto.

4. Você não pode copiar, modificar, sub-licenciar ou distribuir o Programa, exceto de acordo com as condições expressas nesta Licença. Qualquer outra tentativa de cópia, modificação, sub-licenciamento ou distribuição do Programa não é válida, e cancelará automaticamente os direitos que lhe foram fornecidos por esta Licença. No entanto, terceiros que de você receberam cópias ou direitos, fornecidos sob os termos desta Licença, não terão suas licenças terminadas, desde que permaneçam em total concordância com ela.
5. Você não é obrigado a aceitar esta Licença já que não a assinou. No entanto, nada mais o dará permissão para modificar ou distribuir o Programa ou trabalhos derivados deste. Estas ações são proibidas por lei, caso você não aceite esta Licença. Desta forma, ao modificar ou distribuir o Programa (ou qualquer trabalho derivado do Programa), você estará indicando sua total aceitação desta Licença para fazê-los, e todos os seus termos e condições para copiar, distribuir ou modificar o Programa, ou trabalhos baseados nele.
6. Cada vez que você redistribuir o Programa (ou qualquer trabalho baseado nele), os recebedores adquirirão automaticamente do licenciador original uma licença para copiar, distribuir ou modificar o Programa, sujeitos a estes termos e condições. Você não poderá impor aos recebedores qualquer outra restrição ao exercício dos direitos então adquiridos. Você não é responsável em garantir a concordância de terceiros a esta Licença.
7. Se, em consequência de decisões judiciais ou alegações de infringimento de patentes ou quaisquer outras razões (não limitadas a assuntos relacionados a patentes), condições forem impostas a você (por ordem judicial, acordos ou outras formas) e que contradigam as condições desta Licença, elas não o livram das condições desta Licença. Se você não puder distribuir de forma a satisfazer simultaneamente suas obrigações para com esta Licença e para com as outras obrigações pertinentes, então como consequência você não poderá distribuir o Programa. Por exemplo, se uma licença de patente não permitirá a redistribuição, livre de "royalties", do Programa, por todos aqueles que receberem cópias direta ou indiretamente de você, então a única forma de você satisfazer a ela e a esta Licença seria a de desistir completamente de distribuir o Programa.

Se qualquer parte desta seção for considerada inválida ou não aplicável em qualquer circunstância particular, o restante da seção se aplica, e a seção como um todo se aplica em outras circunstâncias.

O propósito desta seção não é o de induzi-lo a infringir quaisquer patentes ou reivindicação de direitos de propriedade outros, ou a contestar a validade de quaisquer dessas reivindicações; esta seção tem como único propósito proteger a integridade dos sistemas de distribuição de software livres, o que é implementado pela prática de licenças públicas. Várias pessoas têm contribuído generosamente e em grande escala para os software distribuídos usando este sistema, na certeza de que sua aplicação é feita de forma consistente; fica a critério do autor/doador decidir se ele ou ela está disposto a distribuir software utilizando outro sistema, e um licenciado não pode impor qualquer escolha.

Esta seção destina-se a tornar bastante claro o que se acredita ser consequência do restante desta Licença.

8. Se a distribuição e/ou uso do Programa são restringidos em certos países por patentes ou direitos autorais, o detentor dos direitos autorais original, e que colocou o Programa sob esta Licença, pode incluir uma limitação geográfica de distribuição, excluindo aqueles países de forma a tornar a distribuição permitida apenas naqueles ou entre aqueles países então não excluídos. Nestes casos, esta Licença incorpora a limitação como se a mesma constasse escrita nesta Licença.
9. A Free Software Foundation pode publicar versões revisadas e/ou novas da Licença Pública Geral de tempos em tempos. Estas novas versões serão similares em espírito à versão atual, mas podem diferir em detalhes que resolvem novos problemas ou situações.

A cada versão é dada um número distinto. Se o Programa especifica um número de versão específico desta Licença que se aplica a ele e a "qualquer nova versão", você tem a opção de aceitar os termos e condições daquela versão ou de qualquer outra versão publicada pela Free Software Foundation. Se o programa não especifica um número de versão desta Licença, você pode escolher qualquer versão já publicada pela Free Software Foundation.

10. Se você pretende incorporar partes do Programa em outros programas livres cujas condições de distribuição são diferentes, escreva ao autor e solicite permissão. Para o software que a Free Software Foundation detém direitos autorais, escreva à Free Software Foundation; às vezes nós permitimos exceções a este caso. Nossa decisão será guiada pelos dois objetivos de preservar a condição de liberdade de todas as derivações do nosso software livre, e de promover o compartilhamento e reutilização de software em aspectos gerais.

AUSÊNCIA DE GARANTIAS

11. UMA VEZ QUE O PROGRAMA É LICENCIADO SEM ÔNUS, NÃO HÁ QUALQUER GARANTIA PARA O PROGRAMA, NA EXTENSÃO PERMITIDA PELAS LEIS APLICÁVEIS EXCETO QUANDO EXPRESSADO DE FORMA ESCRITA, OS DETENTORES DOS DIREITOS AUTORAIS E/OU TERCEIROS DISPONIBILIZAM O PROGRAMA "NO ESTADO", SEM QUALQUER TIPO DE GARANTIAS, EXPRESSAS OU IMPLÍCITAS, INCLUINDO, MAS NÃO LIMITADO A, AS GARANTIAS IMPLÍCITAS DE COMERCIALIZAÇÃO E AS DE ADEQUAÇÃO A QUALQUER PROPÓSITO. O RISCO TOTAL COM A QUALIDADE E DESEMPENHO DO PROGRAMA É SEU. SE O PRO-

GRAMA SE MOSTRAR DEFEITUOSO, VOCÊ ASSUME OS CUSTOS DE TODAS AS MANUTENÇÕES, REPAROS E CORREÇÕES.

12. EM NENHUMA OCASIÃO, A MENOS QUE EXIGIDO PELAS LEIS APLICÁVEIS OU ACORDO ESCRITO, OS DETENTORES DOS DIREITOS AUTORAIS, OU QUALQUER OUTRA PARTE QUE POSSA MODIFICAR E/OU REDISTRIBUIR O PROGRAMA CONFORME PERMITIDO ACIMA, SERÃO RESPONSABILIZADOS POR VOCÊ POR DANOS, INCLUINDO QUALQUER DANO EM GERAL, ESPECIAL, ACIDENTAL OU CONSEQÜENTE, RESULTANTES DO USO OU INCAPACIDADE DE USO DO PROGRAMA (INCLUINDO, MAS NÃO LIMITADO A, A PERDA DE DADOS OU DADOS TORNADOS INCORRETOS, OU PERDAS SOFRIDAS POR VOCÊ OU POR OUTRAS PARTES, OU FALHAS DO PROGRAMA AO OPERAR COM QUALQUER OUTRO PROGRAMA), MESMO QUE TAL DETENTOR OU PARTE TENHAM SIDO AVISADOS DA POSSIBILIDADE DE TAIS DANOS.
- FIM DOS TERMOS E CONDIÇÕES

M.3 Como aplicar estes termos aos seus novos programas

Se você desenvolver um novo programa, e quer que ele seja utilizado amplamente pelo público, a melhor forma de alcançar este objetivo é torná-lo software livre que qualquer um pode redistribuir e alterar, sob estes termos.

Para isso, anexe os seguintes avisos ao programa. É mais seguro anexá-los logo no início de cada arquivo-fonte para reforçarem mais efetivamente a inexistência de garantias; e cada arquivo deve possuir pelo menos a linha de "copyright" e uma indicação de onde o texto completo se encontra.

<uma linha que forneça o nome do programa e uma idéia do que ele faz.> Copyright (C) <ano> <nome do autor> Este programa é software livre; você pode redistribuí-lo e/ou modificá-lo sob os termos da Licença Pública Geral GNU, conforme publicada pela Free Software Foundation; tanto a versão 2 da Licença como (a seu critério) qualquer versão mais nova.

Este programa é distribuído na expectativa de ser útil, mas SEM QUALQUER GARANTIA; sem mesmo a garantia implícita de COMERCIALIZAÇÃO ou de ADEQUAÇÃO A QUALQUER PROPÓSITO EM PARTICULAR. Consulte a Licença Pública Geral GNU para obter mais detalhes. Você deve ter recebido uma cópia da Licença Pública Geral GNU junto com este programa; se não, escreva para a Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA.

Inclua também informações sobre como contactá-lo eletronicamente e por carta.

Se o programa é interativo, faça-o mostrar um aviso breve como este, ao iniciar um modo interativo:

Gnomovision versão 69, Copyright (C) ano nome do autor O Gnomovision não possui QUALQUER GARANTIA; para obter mais detalhes digite 'show w'. Ele é software livre e você está convidado a redistribuí-lo sob certas condições; digite 'show c' para obter detalhes.

Os comandos hipotéticos 'show w' e 'show c' devem mostrar as partes apropriadas da Licença Pública Geral. Claro, os comandos que você usar podem ser ativados de outra forma que 'show w' e 'show c'; eles podem até ser cliques do mouse ou itens de um menu – o que melhor se adequar ao programa.

Você também deve obter do seu empregador (se você trabalha como programador) ou escola, se houver, uma "declaração de ausência de direitos autorais" sobre o programa, se necessário. Aqui está um exemplo; altere os nomes:

Yoyodyne, Inc., aqui declara a ausência de quaisquer direitos autorais sobre o programa 'Gnomovision' (que executa interpretações em compiladores) escrito por James Hacker.

<assinatura de Ty Coon>, 1o. de abril de 1989 Ty Con, Vice-presidente

Esta Licença Pública Geral não permite incorporar seu programa em programas proprietários. Se seu programa é uma biblioteca de sub-rotinas, você deve considerar mais útil permitir ligar aplicações proprietárias com a biblioteca. Se isto é o que você deseja, use a Licença Pública Geral de Bibliotecas GNU, ao invés desta Licença.