

# Treinamento em Linguagem C++

## APOSTILA

(versão 2018-10)

*Basilio Miranda*

---

## **Treinamento em Linguagem C++**

Autor: Basilio Miranda.

Copyright © AGIT Informática Ltda (todos os direitos reservados).

**Esta Apostila está registrada sob o número 298.397, livro 542, folha 57, no Escritório de Direitos Autorais da Fundação Biblioteca Nacional, Ministério da Cultura.**

⇔ **Nenhuma parte desta publicação poderá ser reproduzida**, guardada pelo sistema “*retrieval*” ou transmitida de qualquer modo ou por qualquer outro meio, seja este eletrônico, mecânico, de fotocópia, de gravação, ou outros, **sem prévia autorização, por escrito, da Agit Informática Ltda, sob pena de responder por perdas e danos, infringindo ao disposto na Legislação pertinente, que assegura a proteção aos Direitos Autorais – LEI nº 9.610, de 19 de fevereiro de 1998 do artigo nº 29.**

**Produção e Editoração: AGIT Informática Ltda.**

---

## • Sumário

---

• Sumário detalhado.....	5
• Capítulo 1   ▪ Introdução: a Linguagem C++.....	16
• Capítulo 2   ▪ Iniciando: o básico em C++.....	39
• Capítulo 3   ▪ Programação: princípios básicos.....	71
• Capítulo 4   ▪ Memória: tipos, declarações e definições.....	97
• Capítulo 5   ▪ Fluxo de processamento.....	131
• Capítulo 6   ▪ Ponteiros e referências.....	186
• Capítulo 7   ▪ Iniciando orientação a objetos.....	208
• Capítulo 8   ▪ Programação genérica.....	264
• Capítulo 9   ▪ A biblioteca padrão.....	273
• Capítulo 10   ▪ Classes e objetos: herança.....	289
• Capítulo 11   ▪ Classes e objetos: Polimorfismo.....	299
• Capítulo 12   ▪ Tratamento de Exceções.....	318
• Capítulo 13   ▪ A biblioteca de tempo Chrono.....	327
• Capítulo 14   ▪ Threads.....	330
• Capítulo 15   ▪ Referência técnica.....	350
• Anexos.....	420



---

## • Sumário detalhado

---

• <b>Capítulo 1</b> • Introdução: a Linguagem C++.....	16
1.1 • Tópicos de destaque neste capítulo.....	17
1.2 • As linguagens C e C++.....	19
1.2.1 • Breve histórico.....	19
1.2.1.1 • Linguagem C.....	19
1.2.1.2 • Linguagem C++.....	19
1.2.2 • Características gerais de C++.....	20
1.2.3 • Diferenças entre C e C++.....	21
1.3 • Biblioteca de C++ e bibliotecas auxiliares.....	22
1.3.1 • Biblioteca padrão de C++.....	22
1.3.2 • Bibliotecas adicionais.....	22
1.3.3 • Bibliotecas adicionais: indispensáveis.....	23
1.4 • Porque aprender C++.....	24
1.4.1 • Base sólida em programação.....	24
1.4.2 • C++ pode ser a linguagem principal.....	24
1.4.3 • Usando outras linguagens.....	24
1.5 • Livros e <i>sites</i> sobre C++.....	25
1.5.1 • Livros para o iniciante.....	25
1.5.2 • Livros para aprofundamento.....	26
1.5.3 • Livros adicionais(abordagens e técnicas).....	26
1.5.4 • Alguns <i>sites</i> úteis sobre C++.....	28
1.6 • Linguagem, Compiladores e Ambientes.....	29
1.6.1 • Compiladores.....	29
1.6.2 • Ambientes de Desenvolvimento Integrado.....	30
1.6.3 • Compiladores comerciais e gratuitos.....	31
1.6.4 • Exemplos de compiladores.....	31
1.6.4.1 • Gratuitos (alguns exemplos).....	31
1.6.4.2 • Comerciais (alguns exemplos).....	31
1.6.4.3 • Outros exemplos.....	32
1.6.5 • Exemplos de ambientes de desenvolvimento.....	32
1.7 • O que veremos no curso.....	34
1.8 • Conclusão.....	35
1.8.1 • Linguagem padronizada.....	35
1.8.2 • Linguagem multi-paradigma.....	35
1.9 • Ferramentas usadas no curso.....	36
1.10 • Convenções usadas nesta apostila.....	37
1.10.1 • Versões de C++.....	37
1.10.2 • Símbolos de atenção ou ênfase.....	37
1.11 • Questões para revisão do Capítulo 1.....	38
• <b>Capítulo 2</b> • Iniciando: o básico em C++.....	39
2.1 • Tópicos de destaque neste capítulo.....	40
2.2 • Exemplos preliminares.....	41

2.2.1	▪ Um programa mínimo.....	41
2.2.2	▪ Um programa um pouco maior.....	43
2.2.3	▪ Blocos de código e chaves.....	45
2.2.4	▪ Gramática básica.....	46
2.3	• Compilando e executando.....	47
2.3.1	▪ Exemplo no <i>Windows</i> .....	47
2.3.2	▪ Exemplo em <i>Unix/Linux</i> .....	51
2.4	• Analisando e corrigindo erros de compilação.....	54
2.5	• Usando Ambientes de Desenvolvimento.....	56
2.5.1	▪ Qt Creator.....	57
2.5.1.1	▪ Criando um projeto.....	57
2.5.1.2	▪ Adicionando arquivos ao projeto.....	61
2.5.1.3	▪ Compilar e executar.....	62
2.5.2	▪ Visual C++.....	63
2.5.2.1	▪ Criando um projeto.....	63
2.5.2.2	▪ Adicionando arquivos ao projeto.....	65
2.5.2.3	▪ Compilar e executar.....	66
2.6	• Conclusões sobre o código usado.....	67
2.7	• Revisão do Capítulo 2.....	68
2.7.1	▪ Exercício.....	68
2.7.2	▪ Questões para revisão.....	69
• Capítulo 3	• Programação: princípios básicos.....	71
3.1	• Tópicos de destaque neste capítulo.....	72
3.2	• Operações, memória e fluxo de processamento.....	73
3.2.1	▪ Operações.....	73
3.2.2	▪ Memória.....	73
3.2.3	▪ Fluxo de processamento.....	74
3.2.4	▪ Conceitos de Verdadeiro e Falso.....	75
3.2.5	▪ Implementação em C++.....	77
3.2.5.1	▪ Operações e operadores.....	77
3.2.5.2	▪ Memória.....	77
3.2.5.3	▪ Escrevendo código.....	78
3.2.5.4	▪ Verdadeiro e Falso.....	79
3.3	• Fluxo de processamento: laços.....	80
3.3.1	▪ Princípios.....	80
3.3.2	▪ Implementação em C++.....	82
3.3.2.1	▪ O controle de laço "while".....	82
3.3.2.2	▪ Compilando e executando.....	86
3.3.2.3	▪ Laço for.....	86
3.3.3	▪ O tipo bool e o resultado de avaliações.....	89
3.4	• Comentários.....	89
3.5	• Conclusão.....	90
3.6	• Revisão do Capítulo 3.....	91
3.6.1	▪ Exercício.....	91
3.6.2	▪ Questões para revisão.....	94
• Capítulo 4	• Memória: tipos, declarações e definições.....	97
4.1	• Tópicos de destaque neste capítulo.....	99
4.2	• Reserva e acesso de memória em C e C++.....	100
4.2.1	▪ Tipos de dados.....	100
4.2.2	▪ Escolhendo o tipo adequado.....	100
4.2.3	▪ Porque reservar memórias com um tipo.....	102
4.2.4	▪ Para lembrar.....	102
4.2.5	▪ Criando sinônimos de tipos (C++98 e C++11).....	103

4.2.5.1 • Para que servem sinônimos de tipos.....	103
4.3 • Inicialização e atribuição.....	103
4.4 • Variáveis e constantes.....	104
4.4.1 • Porque associar constantes a um nome.....	104
4.4.2 • Declarando constantes com <i>enum</i> .....	105
4.4.3 • Enum: diferenças entre C e C++ e entre C++98 e C++11.....	106
4.4.3.1 • Denominação de um tipo criado com enum.....	106
4.4.3.2 • Enum sem e com escopo próprio.....	107
4.4.3.3 • Definindo o tipo inteiro subjacente de um enum (C++11).....	107
4.4.3.4 • Exemplos de escopo e especificação do seu tipo inteiro.....	107
4.4.4 • Constantes nomeadas em um único lugar.....	108
4.5 • Declarações auto (C++11).....	109
4.6 • Conceitos: valores, tipos e objetos.....	109
4.7 • Conversões entre tipos.....	110
4.8 • Variáveis classificadas como <i>static</i> e <i>extern</i> .....	113
4.8.1 • Variáveis <i>static</i> de um bloco.....	113
4.8.2 • Variáveis <i>static</i> de um módulo e variáveis <i>extern</i> .....	114
4.8.3 • Conformidade com C++.....	115
4.9 • Como a memória é organizada.....	115
4.9.1 • Memória global ou estática.....	115
4.9.2 • A pilha ( <i>stack</i> ).....	115
4.9.3 • Livre alocação.....	115
4.10 • Vetores.....	116
4.10.1 • Vetores: o que são e para que servem.....	116
4.10.2 • <i>std::string</i> e <i>std::vector</i> .....	116
4.10.3 • Vetores na linguagem C.....	119
4.10.4 • Usos seguros de índices em vetores.....	122
4.10.5 • Caracteres: aspas simples e duplas.....	123
4.11 • Revisão do Capítulo 4.....	124
4.11.1 • Exercício.....	124
4.11.2 • Questões para revisão.....	128
• Capítulo 5 • Fluxo de processamento.....	131
5.1 • Tópicos de destaque neste capítulo.....	133
5.2 • Linhas de instrução; declarações e operações.....	135
5.3 • Operadores.....	136
5.3.1 • Acrescentando mais alguns operadores.....	136
5.3.2 • Operador condicional ternário.....	137
5.3.3 • Resto de divisão.....	138
5.3.4 • Operadores compostos.....	139
5.3.5 • Operações lógicas.....	141
5.3.6 • Em conclusão.....	143
5.4 • Precedência de operadores.....	143
5.5 • Finalização de uma linha de instrução.....	144
5.6 • Funções.....	144
5.6.1 • Parâmetros e valor de retorno de uma função.....	145
5.6.2 • Funções sem parâmetros e/ou valor de retorno.....	146
5.6.3 • A função <i>main</i> .....	147
5.6.3.1 • Retorno de <i>main</i> .....	148
5.6.3.2 • Parâmetros de <i>main</i> .....	149
5.6.4 • Protótipos de funções.....	150
5.6.5 • Arquivos <i>header</i> .....	151
5.6.6 • Escopo de uma função e escopo global.....	153

5.6.7	Chamadas de função e seu custo.....	154
5.6.8	Funções inline.....	154
5.6.9	Funções recursivas.....	156
5.6.10	Funções sobrecarregadas.....	158
5.7	Declarações <i>constexpr</i> (em C++11 e em C++14).....	158
5.7.1	Funções <i>constexpr</i> .....	158
5.7.1.1	Em C++11.....	158
5.7.1.2	Em C++14.....	159
5.7.2	Objetos <i>constexpr</i> e diferença para <i>const</i> .....	159
5.8	Lambdas (C++11).....	160
5.8.1	Expressão lambda através de uma variável auto.....	161
5.8.2	Capturando variáveis em lambdas.....	162
5.8.3	Generic lambdas (C++ 14).....	162
5.9	Tomadas de decisão (detalhamento).....	163
5.9.1	if / else.....	163
5.9.1.1	Cuidados a tomar com o if.....	163
5.9.1.2	A precaução com a atribuição também se aplica aos laços.....	167
5.9.2	switch.....	167
5.10	Laços.....	168
5.10.1	Porque usar laços estruturados.....	168
5.10.2	Laço do ... while.....	169
5.10.3	Desvios por salto incondicional.....	170
5.10.3.1	Desvio <i>return</i> .....	170
5.10.3.2	Desvio <i>break</i> .....	170
5.10.3.3	Desvio <i>continue</i> .....	171
5.10.3.4	Desvio <i>goto</i> .....	174
5.11	Revisão do Capítulo 5.....	174
5.11.1	Sintetizando as regras básicas para escrita de código.....	174
5.11.2	Exercício 1.....	175
5.11.2.1	Iniciando o exercício.....	176
5.11.3	Exercício 2.....	179
5.11.4	Questões para revisão.....	180
<b>Capítulo 6</b>	<b>Ponteiros e referências.....</b>	<b>186</b>
6.1	O problema do carteiro.....	187
6.2	Trabalhando com endereços.....	189
6.2.1	Exemplo com ponteiros.....	190
6.2.2	Ponteiros nulos (nullptr em C++11).....	191
6.2.3	Ponteiros para Funções.....	192
6.2.3.1	Definindo os tipos dos ponteiros para função.....	193
6.2.3.2	Ponteiros para função permitem estabelecer eventos.....	194
6.2.4	Usando ponteiros como parâmetros de funções.....	195
6.2.5	Evitando duplos acessos de memória.....	198
6.3	Referências (lvalue).....	200
6.3.1	Inicialização de referências.....	201
6.3.2	Diferenciando referências de ponteiros.....	201
6.3.3	Exemplo com referências (e parâmetros por referência).....	202
6.4	Referências (rvalue - C++11).....	204
6.4.1	Semântica move (C++11).....	205
6.5	Questões para revisão do Capítulo 6.....	206
<b>Capítulo 7</b>	<b>Iniciando orientação a objetos.....</b>	<b>208</b>
7.1	Estruturas.....	210
7.1.1	Estruturas e funções relacionadas.....	212



7.1.2	▪ Limitação da linguagem C.....	212
7.1.3	▪ Passando estruturas como argumento.....	214
7.1.4	▪ A especificação <i>const</i> .....	217
7.2	• Estruturas em C++ : encapsulamento.....	217
7.3	• Implementando as funções membras.....	219
7.4	• As palavras reservadas <i>struct</i> e <i>class</i> .....	221
7.4.1	▪ Definindo o objeto apontado por “ <i>this</i> ” como <i>const</i> .....	222
7.4.2	▪ Implementação das funções membras <i>const</i> .....	223
7.4.3	▪ Testando a <i>class Data</i> (testadata.cpp).....	229
7.4.4	▪ Especificando funções como <i>inline</i> .....	230
7.4.5	▪ Usando um segundo parâmetro do tipo “ <i>Data</i> ”.....	231
7.4.6	▪ Acrescentando funções operadoras à classe “ <i>Data</i> ”.....	233
7.4.7	▪ Os operadores << e >> de ostream/istream.....	235
7.4.8	▪ Conclusão Parcial.....	236
7.5	• Encapsulamento: reforçando conceitos.....	237
7.5.1	▪ Funções inline.....	237
7.5.2	▪ O ponteiro <i>this</i> .....	237
7.5.2.1	▪ Definindo o objeto apontado por <i>this</i> como <i>const</i> .....	238
7.5.2.2	▪ Exceção para a restrição <i>const</i> de <i>this</i> ( <i>mutable</i> ).....	239
7.5.3	▪ Função Construtora.....	239
7.5.3.1	▪ Parâmetros para a função construtora.....	240
7.5.3.2	▪ Contrutora <i>default</i> .....	240
7.5.3.3	▪ Construtora de cópia e operador de atribuição.....	241
7.5.3.4	▪ Construtoras com conversão implícita e explícita.....	242
7.5.3.5	▪ Delegando construtoras (C++11).....	243
7.5.4	▪ Função destrutora.....	244
7.5.5	▪ Alterando o ponto de entrada da aplicação: objetos externos.....	245
7.5.6	▪ Classes e funções amigas.....	245
7.5.7	▪ Membros estáticos de uma classe.....	246
7.5.7.1	▪ Membros de dados estáticos.....	246
7.5.7.2	▪ Funções-membro estáticas.....	248
7.5.8	▪ Objetos como membros de classes.....	249
7.5.8.1	▪ Membros “objeto de outra classe”.....	249
7.5.8.2	▪ Membros “ponteiro para objeto de outra classe”.....	250
7.5.8.3	▪ Membros “ponteiro para objeto da mesma classe”.....	251
7.5.9	▪ Literais definidos pelo usuário (C++11).....	252
7.5.10	▪ Ambientes de nomes.....	253
7.5.10.1	▪ Evitando colisões de nomes.....	253
7.5.10.2	▪ Usando namespace para organizar conjuntos de software.....	254
7.5.10.3	▪ O namespace “ <i>std</i> ”.....	255
7.5.10.4	▪ Namespace inline (C++11).....	256
7.5.10.5	▪ Exemplos de uso de namespace.....	257
7.6	• Questões para revisão do capítulo 7.....	260
• Capítulo 8	▪ Programação genérica.....	264
8.1	• Templates.....	265
8.2	• Templates de função.....	265
8.3	• Templates de classes.....	268
8.4	• Templates com quantidade variável de argumentos (C++11).....	270
• Capítulo 9	▪ A biblioteca padrão.....	273
9.1	• <i>std::string</i> .....	276
9.2	• <i>std::vector</i> .....	277

9.3 • std::list.....	277
9.4 • std::map.....	279
9.5 • std::vector < std::vector < > >.....	280
9.6 • Laço <i>for</i> para <i>containers</i> (C++11).....	281
9.7 • Outras classes containers.....	282
9.7.1 • <i>initializer_list</i> (C++11).....	282
9.7.2 • <i>forward_list</i> (C++11).....	283
9.7.3 • <i>unordered_map</i> (C++11).....	285
9.7.4 • <i>unordored_multimap</i> (C++11).....	285
9.7.5 • <i>unordered_set</i> (C++11).....	286
9.7.6 • <i>unordered_multiset</i> (C++11).....	286
9.7.7 • <i>array</i> (C++11).....	287
9.7.8 • <i>tuple</i> (C++11).....	287
<b>• Capítulo 10 • Classes e objetos: herança.....</b>	<b>289</b>
10.1 • Regras básicas.....	290
10.1.1 • Restrição de acesso <i>protected</i> .....	290
10.1.2 • Modos de derivação.....	290
10.1.2.1 • Derivação pública ( : public).....	290
10.1.2.2 • Derivação privada ( : private).....	290
10.1.2.3 • Derivação protegida ( : protected).....	290
10.2 • Quando usar herança.....	291
10.3 • Construtoras e destrutoras na derivada e na base.....	292
10.3.1 • Criação de um objeto de uma classe derivada.....	292
10.3.2 • Destruição de um objeto de uma classe derivada.....	292
10.3.3 • Como a construtora derivada pode chamar uma construtora base.....	292
10.3.4 • Herdando construtoras (C++11).....	293
10.4 • Herança Múltipla.....	295
10.4.1 • Herança múltipla e herança virtual:.....	296
<b>• Capítulo 11 • Classes e objetos: Polimorfismo.....</b>	<b>299</b>
11.1 • Sobrecarga.....	300
11.1.1.1 • Sobrecarga de funções.....	300
11.1.1.2 • Sobrecarga de operadores.....	300
11.2 • Redefinição de funções nas classes derivadas.....	302
11.3 • Funções virtuais (polimorfismo estrito).....	304
11.3.1 • Polimorfismo em <i>frameworks</i> : ponteiros para função e funções virtuais.....	305
11.3.2 • Declarar e implementar funções virtuais.....	306
11.3.3 • Final e override (C++11).....	308
11.3.3.1 • Final (C++11).....	308
11.3.3.2 • Override (C++11).....	309
11.3.4 • Funções virtuais como respostas a eventos.....	309
11.3.4.1 • Funções virtuais puras e classes abstratas.....	312
11.3.4.2 • Exercício: a classe <i>RelatPadrao</i> .....	312
11.3.4.2.a • Solução: declaração, implementação e uso da classe <i>RelatPadrao</i> .....	313
11.3.5 • Precauções ao usar funções virtuais.....	314
11.3.5.1.a • 1 - Nunca preencher um objeto com métodos de baixo nível.....	314
11.3.5.1.b • 2 - Nunca esquecer como a <i>vtable</i> é preenchida.....	314
11.4 • Questões para revisão.....	315

• <b>Capítulo 12</b> • Tratamento de Exceções.....	318
12.1 • Usando throw ... try {...} catch(...) {...}.....	319
12.2 • Usando um tratador <i>default</i> .....	320
12.3 • <code>std::exception</code> .....	321
12.4 • Derivadas padrão de <code>std::exception</code> .....	324
• <b>Capítulo 13</b> • A biblioteca de tempo Chrono.....	327
• <b>Capítulo 14</b> • Threads.....	330
14.1 • A classe <code>std::thread</code> (C++11).....	331
14.2 • Async Thread (C++11).....	332
14.3 • Mutex (C++11).....	334
14.4 • Recursive mutex (C++11).....	336
14.5 • Timed mutex (C++11).....	336
14.6 • Recursive timed mutex (C++11).....	337
14.7 • Lock guard (C++11).....	338
14.8 • Unique lock (C++11).....	339
14.9 • Future/Promise (C++11).....	339
14.10 • Call once (C++11).....	342
14.11 • TLS – Thread Local Storage (C++11).....	343
14.12 • Atomic (C++11).....	343
• <b>Capítulo 15</b> • Referência técnica.....	350
15.1 • Conversões de tipos.....	352
15.1.1 • Conversões no estilo C.....	352
15.1.2 • Conversões no estilo C++.....	353
15.2 • Tipos agregados e aninhados.....	355
15.2.1 • Tipos aninhados .....	356
15.2.2 • Estruturas aninhadas.....	356
15.2.2.1 • Tipos aninhados: em C eles são apenas indicativos.....	357
15.2.2.2 • Tipos aninhados: C++ não permite o estilo C.....	357
15.2.2.3 • Criação de tipos diretamente no retorno ou nos parâmetros de uma função.....	357
15.3 • Criando tipos através de <i>union</i> .....	358
15.3.1 • Regras para <i>union</i> , específicas de C++.....	359
15.3.2 • Unions com menos restrições (C++11).....	359
15.3.2.1 • Inicialização uniforme (C++11).....	362
15.3.3 • Resultados de operações.....	363
15.3.3.1 • Operações relacionais.....	363
15.3.3.2 • Operações lógicas.....	363
15.3.3.3 • Operações aritméticas.....	363
15.3.4 • Posição dos operadores de incremento e decremento.....	364
15.3.5 • Operadores <i>bit a bit</i> .....	364
15.3.5.1 • AND.....	365
15.3.5.2 • OR.....	365
15.3.5.3 • NOT (ou complemento de 1).....	365
15.3.5.4 • XOR (ou exclusivo).....	366
15.3.5.5 • Shift (ou deslocamento) à esquerda.....	366
15.3.5.6 • Shift (ou deslocamento) à direita.....	366
15.3.5.7 • Exemplos com operadores de <i>bits</i> .....	367
15.3.5.7.a • Exemplo com <i>and</i> , <i>or</i> , e <i>not</i> .....	367
15.3.5.7.b • Exemplo com <i>shift</i> (à esquerda e à direita).....	369
15.3.5.7.c • Exemplo com <i>xor</i> (1).....	370
15.3.5.7.d • Exemplo com <i>xor</i> (2).....	372

15.4 • Controles do fluxo de processamento.....	375
15.4.1.1 • Parâmetros para funções e retorno de funções.....	375
15.4.1.2 • Declarando parâmetros.....	375
15.4.1.3 • Formas de declaração.....	375
15.4.1.3.a • Forma de declaração válida em C e C++.....	375
15.4.1.3.b • Forma de declaração inválida em C++.....	375
15.4.1.4 • Funções com quantidade fixa de parâmetros.....	375
15.4.1.5 • Funções com quantidade variável de parâmetros.....	376
15.4.1.5.a • Declarando os parâmetros desconhecidos.....	376
15.4.1.5.b • Exemplo de função com quantidade de parâmetros variável.....	377
15.4.1.6 • Valor de retorno, tipo e protótipo de funções.....	377
15.4.1.6.a • Valor de retorno.....	377
15.4.1.6.b • Tipo de uma função.....	378
15.4.1.6.c • Protótipo de funções.....	379
15.4.1.7 • Modos de passagem de parâmetros (recapitulando).....	380
15.4.1.7.a • 1 - Passando parâmetros por valor.....	380
15.4.1.7.b • 2 - Passando parâmetros por endereço.....	381
15.4.1.7.c • 3 - Passando parâmetros por referência.....	384
15.4.1.8 • Há dois motivos para passar parâmetros por endereço ou por referência.....	385
15.4.1.9 • Impedindo efeitos colaterais desnecessários.....	386
15.4.2 • Lógica estruturada através de objetos.....	386
15.4.3 • Controles de laço (recapitulando).....	387
15.4.3.1 • O laço <i>for</i> .....	387
15.4.3.1.a • Forma geral e funcionamento.....	387
15.4.3.1.b • Sintaxe.....	388
15.4.3.1.c • Cuidados a tomar com o laço <i>for</i> .....	390
15.4.3.2 • O laço <i>while</i> .....	391
15.4.3.2.a • Forma geral e funcionamento.....	391
15.4.3.2.b • Sintaxe.....	391
15.4.3.2.c • Cuidados a tomar com o laço <i>while</i> .....	392
15.4.3.3 • Considerações gerais sobre os laços <i>for</i> e <i>while</i> ( <i>tenha cuidado com...</i> ).....	394
15.4.3.4 • O laço <i>do ... while</i> .....	394
15.4.3.4.a • Forma geral, funcionamento e sintaxe.....	394
15.5 • Ponteiros, Matrizes e Referências.....	397
15.5.1.1 • Alocando livremente memória no <i>heap</i> (evite...).....	397
15.5.1.2 • Acessando valores através de ponteiros.....	398
15.5.1.3 • Exercícios: demonstrando o funcionamento de ponteiros.....	398
15.5.2 • Ponteiros inteligentes (smart pointers).....	401
15.5.2.1 • Auto ptr (obsoleto a partir de C++11).....	403
15.5.2.2 • unique_ptr (C++11).....	403
15.5.2.3 • shared_ptr (C++11).....	404
15.5.2.4 • weak_ptr (C++11).....	405
15.5.3 • Vetores e Matrizes.....	406
15.5.3.1 • Onde alocar vetores: memória global, pilha ou <i>heap</i> ?.....	407
15.5.3.2 • Matrizes - usando múltiplas dimensões.....	409
15.5.3.3 • Inicialização de vetores e matrizes.....	410
15.5.3.4 • Vetores e Matrizes de estruturas.....	410
15.5.3.4.a • Inicializando uma matriz de estruturas.....	411
15.5.3.5 • Vetores e Matrizes de caracteres.....	412
15.5.3.5.a • Inicializando uma matriz de caracteres.....	413
15.5.3.5.b • Entendendo matriz de caracteres como um endereço.....	413
15.5.3.5.c • Exemplo de matriz de caracteres.....	414
15.6 • Ponteiros para funções(exemplo adicional).....	416
A estrutura relatório.....	416

• Anexos.....	420
<b>Anexo A.</b> Compilador e <i>linker</i> .....	422
<b>Anexo B.</b> Respostas às questões para revisão.....	424
1. Questões do Capítulo 1.....	425
2. Questões do Capítulo 2.....	427
1. Exercício.....	427
2. Questões.....	428
3. Questões do Capítulo 3.....	431
1. Exercício.....	431
2. Questões.....	433
4. Questões do Capítulo 4.....	436
1. Exercício.....	436
2. Questões.....	440
5. Questões do Capítulo 5.....	446
1. Exercício 1.....	446
2. Exercício 2.....	452
3. Questões.....	455
6. Questões do Capítulo 6.....	468
7. Questões do Capítulo 7.....	471
8. Questões do Capítulo 11.....	473
<b>Anexo C.</b> Guia de consulta rápida.....	476
1. Tabela de tipos primitivos.....	477
2. Tabela de operadores <i>básicos</i> por tipo de operação.....	479
3. Tabela completa de operadores e suas precedências.....	481
4. Controles de fluxo de processamento.....	483
5. Principais diretivas de compilação.....	485
6. sequências <i>escape</i> .....	486
7. Palavras reservadas.....	486
8. Modificadores.....	487

"

Nunca projete (*um software*) sozinho se você puder evitar isso! Não comece a escrever código antes que você tenha colocado suas idéias a prova, explicando-as para alguém. Discuta o projeto, *designs*, e técnicas de programação com amigos, colegas, usuários potenciais, etc, antes de começar a usar o teclado. É incrível o quanto você pode aprender apenas experimentando articular as idéias. No fim das contas, um programa nada mais é do que a expressão (em código) de determinadas idéias.

(...)

Do mesmo modo, quando você já estiver implementando um programa, e os erros aparecerem, tire os olhos do teclado. Pense no problema em si, ao invés de se fixar na solução - ainda incompleta- que você desenvolveu. Converse com alguém: explique o que você quer fazer e porque o que você já fez não está funcionando. É surpreendente o quanto é comum que uma solução seja encontrada apenas porque explicamos cuidadosamente o problema para alguém. Não procure descobrir erros sozinho se você não for obrigado a fazer isso!

(...)

Quando você buscar uma abordagem para a criação de *software*, faça isso de uma maneira fundamentada e séria: você quer ser parte da solução e não mais um dos problemas.

"

**Bjarne Stroustrup**, criador original da Linguagem C++,  
no livro "**Programming - Principles and Practice Using C++**".



# • Capítulo 1

## ▪ Introdução: a Linguagem C++

Neste capítulo teremos uma visão geral sobre:

- A linguagem **C++** e suas principais características.
- A importância de bibliotecas que fornecem código já pronto para uso.
- As ferramentas necessárias para gerar programas executáveis.
- Recursos auxiliares para o aprendizado de **C++**, como livros e *sites*.

1.1 • Tópicos de destaque neste capítulo.....	17
1.2 • As linguagens C e C++.....	19
1.2.1 • Breve histórico.....	19
1.2.1.1 • Linguagem C.....	19
1.2.1.2 • Linguagem C++.....	19
1.2.2 • Características gerais de C++.....	20
1.2.3 • Diferenças entre C e C++.....	21
1.3 • Biblioteca de C++ e bibliotecas auxiliares.....	22
1.3.1 • Biblioteca padrão de C++.....	22
1.3.2 • Bibliotecas adicionais.....	22
1.3.3 • Bibliotecas adicionais: indispensáveis.....	23
1.4 • Porque aprender C++.....	24
1.4.1 • Base sólida em programação.....	24
1.4.2 • C++ pode ser a linguagem principal.....	24
1.4.3 • Usando outras linguagens.....	24
1.5 • Livros e <i>sites</i> sobre C++.....	25
1.5.1 • Livros para o iniciante.....	25
1.5.2 • Livros para aprofundamento.....	26
1.5.3 • Livros adicionais(abordagens e técnicas).....	26
1.5.4 • Alguns <i>sites</i> úteis sobre C++.....	28
1.6 • Linguagem, Compiladores e Ambientes.....	29
1.6.1 • Compiladores.....	29
1.6.2 • Ambientes de Desenvolvimento Integrado.....	30
1.6.3 • Compiladores comerciais e gratuitos.....	31
1.6.4 • Exemplos de compiladores.....	31
1.6.4.1 • Gratuitos (alguns exemplos).....	31
1.6.4.2 • Comerciais (alguns exemplos).....	31
1.6.4.3 • Outros exemplos.....	32
1.6.5 • Exemplos de ambientes de desenvolvimento.....	32
1.7 • O que veremos no curso.....	34
1.8 • Conclusão.....	35
1.8.1 • Linguagem padronizada.....	35
1.8.2 • Linguagem multi-paradigma.....	35
1.9 • Ferramentas usadas no curso.....	36
1.10 • Convenções usadas nesta apostila.....	37
1.10.1 • Versões de C++.....	37
1.10.2 • Símbolos de atenção ou ênfase.....	37



1.11 • Questões para revisão do Capítulo 1.....	38
---	----

## 1.1 • Tópicos de destaque neste capítulo

### ➤ C e C++ são linguagens *padronizadas* (ISO/ANSI).

- ◆ São padrões abertos, disponíveis em qualquer plataforma computacional.
- ◆ Por isso são linguagens multiplataforma, exigindo apenas uma nova *compilação* para cada plataforma desejada.

### ➤ Para poder utilizar C++ em *diferentes plataformas* precisamos de:

- ◆ *Compiladores* que traduzam o código fonte para o código de máquina de cada plataforma.
- ◆ E, em praticamente todas as plataformas, encontramos compiladores C++, tanto comerciais como gratuitos.
- ◆ Encontramos também *ambientes de desenvolvimento*, comerciais e gratuitos.

### ➤ C++ é, quase totalmente, *compatível* com C.

### ➤ Mas os novos recursos oferecidos por C++ conduzem a uma *nova maneira* de conceber a programação de computadores.

- ◆ Entre esses novos recursos destacam-se duas vertentes principais: *orientação a objetos* e *programação genérica*.
- ◆ Isso significa também que C++ é uma linguagem híbrida, pois, ao contrário de outras, que suportam apenas uma única técnica de programação (como *orientação a objetos*, por exemplo), C++ suporta múltiplas técnicas de programação.  
Por isso dizemos que C++ é uma linguagem *multi-paradigma*.

### ➤ C++ conta com uma *biblioteca padrão*. E há também diversas *outras bibliotecas* que podem ser usadas.

- ◆ A *biblioteca padrão* faz parte do padrão da linguagem e nos oferece, prontos para uso, recursos de suporte em certas áreas de desenvolvimento previsíveis e comuns.
- ◆ Além disso existem diversas bibliotecas adicionais, desenvolvidas por terceiros e consagradas pelo uso, e, neste aspecto, destacamos duas delas: *boost* e *Qt*.
- ◆ Para o programador, isso representa um ganho significativo de *produtividade*.

➤ **Aprender C++ é essencial para um bom programador de computadores.**

- ♦ Porque essa é uma linguagem que poderá ser utilizada em uma grande variedade de situações.
- ♦ E também porque, devido à sua amplitude, já que suporta diversas técnicas de programação, o aprendizado de outras linguagens torna-se mais simples.  
E, aliás, muitas delas são descendentes de C++.

➤ **Escolher C++ como linguagem principal não impede que outras linguagens sejam utilizadas, conjuntamente ou não, sempre que um determinado projeto torne isso mais produtivo ou eficiente.**

➤ **O aprendizado de C++ é atualmente facilitado pela existência de inúmeros sites e livros de excelente nível.**

➤ **Apenas em plataformas muito “baixas” (microcontroladoras, por exemplo), não encontramos compiladores C++. Mas sempre encontraremos um compilador C.**

- ♦ Desse modo é importante que o programador C++ saiba como programar em C.
- ♦ Além disso, o programador C++, em certas situações, terá que lidar com código em C (por exemplo, para usar recursos nativos de sistemas operacionais).
- ♦ Este é um curso sobre C++. Por isso o uso de C é apresentado em segundo lugar, exceto naquilo que é básico e comum às duas linguagens.

## 1.2 • As linguagens C e C++

### 1.2.1 • Breve histórico

#### 1.2.1.1 • Linguagem C

A **linguagem C** foi criada no início da década de 70 por Dennis Ritchie a partir da linguagem **B** (criada por Ken Thompson) que, por sua vez, era baseada na linguagem **BCPL** (criada por Martin Richards).

A popularização de **C** teve grande impulso com o livro “*C – A linguagem de Programação*” de Dennis Ritchie e Brian Kernighan.

Desenvolvida inicialmente no sistema operacional UNIX, a linguagem passou por uma grande expansão, e, ainda na década de 70, surgiram novos compiladores, inclusive para *mainframes* como o IBM 370 e o Honeywell Bull 6000.

Em 1983, o ANSI (*American National Standards Organization* – Instituto Nacional Americano de Padrões) criou um comitê para a padronização da linguagem, visando garantir que os diversos compiladores, então já disponíveis e fornecidos por diferentes empresas, não viessem a adotar caminhos próprios, o que geraria incompatibilidades entre eles.

Assim temos o padrão **C89** (referente a 1989) e o **C99** (referente a 1999), embora, provavelmente, o mais usado ainda seja o **C89**.

#### 1.2.1.2 • Linguagem C++

A **linguagem C++** foi criada entre 1979 e 1983 por Bjarne Stroustrup (autor de “*The C++ Programming Language*”, atualmente em sua quarta edição).

Essa trabalho foi realizado nos “*Bell Laboratories*” que na época pertenciam à empresa **AT&T** (*American Telephone and Telegraph*), onde Stroustrup trabalhava. O objetivo era desenvolver uma linguagem que tivesse a *performance* e flexibilidade da linguagem **C**, mas incorporasse recursos importantes de outras linguagens, especialmente **Simula** (mas também foram utilizadas contribuições originárias de **ADA**, **Algol** e **Clu**).

A **AT&T** tornou esse projeto público o que viria a permitir a criação de um padrão aberto.

A partir dessa liberação um grande número de profissionais de diferentes empresas iniciaram o trabalho pela padronização da Linguagem **C++**, o que levou à criação de um comitê (o comitê **X3J16**) do *American National Standards Institute* (**ANSI**) que teve seus trabalhos iniciados em dezembro de **1989**.

E, em junho de **1991**, esse esforço foi ampliado com o envolvimento direto do *International Standards Organization* (**ISO**) no projeto. A partir daí a iniciativa **ANSI** tornou-se parte da iniciativa **ISO**, integrando-se ao comitê **ISO WG21**.

Em 1995 os comitês **X3J16** e **WG21** publicaram o primeiro documento oficial de padronização de **C++**.

Em 1997 (refletindo o trabalho efetuado em 1996) tivemos um segundo documento intitulado “*1997 C++ Public Review Document*” disponível em versão “html” gratuita nos sites:

<http://www.open-std.org/jtc1/sc22/open/n2356/> <http://anubis.dkuug.dk/jtc1/sc22/open/n2356/>.



E finalmente, em **1998** (refletindo o trabalho efetuado em 1997), foi publicado o último documento oficial de padronização da linguagem, sendo este, até 2003, o padrão oficial de **C++**, o qual pode ser adquirido diretamente no *site* <http://www.ansi.org>.

Desse modo, o padrão inicial de C++ é denominado "**C++98**", o qual recebeu, em **2003**, pequenas alterações (denominadas "**technical corrigendum**"), as quais levaram ao padrão "**C++03**".

Já em 2011 foi lançada a versão **C++11** com grandes modificações em relação à versão anterior, incluindo até mesmo suporte a *threads* e *lambdas*. Em 2014, a versão **C++14** foi lançada com algumas modificações em relação ao **C++11**, mas nada muito drástico. E em 2017 a versão **C++17** foi lançada com várias modificações, inclusive, na biblioteca, suporte ao sistema de arquivos baseado na biblioteca *boost*.

**Esta apostila já inclui os principais recursos de C++11, 14 e 17.**

Informações relevantes sobre a criação e a evolução de C++ podem ser obtidas também no *site* do **criador da linguagem**, Bjarne Stroustrup:

<http://www.stroustrup.com/>.

## 1.2.2 • Características gerais de C++

A linguagem C++ é baseada principalmente em C, mas inspirou-se também em outras linguagens, como **Simula** (especialmente), além de **ADA**, **Algol** e **CLU**, aproveitando destas últimas idéias que, em alguma medida, relacionavam-se a conceitos de **programação orientada a objetos** (OOP), ou a conceitos de **programação genérica**.

Assim, a linguagem C++ nasceu para incorporar nativamente novas técnicas de programação, mas com o **compromisso** de não perder **performance** com relação ao C, mantendo seja a sua flexibilidade, seja os seus recursos de “baixo nível”, que nos permitem resolver certos problemas críticos.

C e C++ são duas linguagens independentes. Ou seja, é perfeitamente possível **aprender C++ sem conhecer C**.

Na prática, contudo, é interessante que o programador C++ conheça as diferenças com relação à linguagem C (e, por consequência, acabe aprendendo como funciona a linguagem C) por, pelo menos, três motivos:

- a. **A linguagem C**, em quase sua totalidade, é um subconjunto de **C++**.
  - Isto significa que **quase tudo** o que é válido em **C** continua sendo válido em **C++** (com **poucas exceções**, que veremos mais adiante).
- b. O programador **C++** encontrará incontáveis bibliotecas já escritas em **C** e que serão úteis (ou mesmo indispensáveis) em muitas e diversas oportunidades.
  - Isso se aplica especialmente ao uso das **APIs** (interfaces de programação de aplicativos) dos sistemas operacionais mais importantes, quase todas elas bibliotecas escritas em **C**, bem como das bibliotecas nativas de sistemas específicos (como bancos de dados e outras).
- c. Em plataformas muito “baixas” (microcontroladoras, por exemplo), nem sempre contaremos com um compilador **C++** e sim, apenas, com um compilador **C** (além, obviamente, do *assembly* da plataforma).
  - Nesses casos, o programador **C++** terá que atuar totalmente em **C**.



**Conclusão:** é importante que o programador **C++** saiba como lidar com códigos escritos em **C**, pois, em algum momento, talvez precise usá-los.

### 1.2.3 • Diferenças entre C e C++

Dissemos que *quase tudo* o que é válido em **C** é válido em **C++**. E que é importante conhecer **C**.



Mas isso não significa que possamos subestimar as diferenças de **C++** com relação a **C**.

Note que os motivos que apresentamos acima para sustentar a posição de que o programador **C++** deve entender o funcionamento de **C**, referem-se, fundamentalmente, ao **aproveitamento de código antigo** (isto é, código escrito anteriormente por você ou por terceiros).

Contudo, no que diz respeito à escrita de **código novo**, o melhor é que você “esqueça” **C** (ainda que isto não seja realmente possível...) e “**raciocine em C++**”.

Exceto, conforme já foi dito, quando precisar programar para plataformas muito baixas, onde só exista um compilador **C**.



Se conseguirmos absorver a **filosofia de C++** seremos conduzidos, na escrita de código novo, a um **estilo de programação bem diferente** do estilo **C mais comum**.

**C** é tão flexível que até é possível escrever programas em **C** com um “**estilo C++ aproximado**”, embora exigindo  **muito mais trabalho**  do programador, já que esta linguagem não suporta diretamente (nativamente) as novas técnicas de programação presentes em **C++**.

Portanto, só entendendo conceitualmente as técnicas e o estilo **próprios** de **C++** é que poderemos aproveitar todas as suas melhorias com relação a **C**.

Existe uma série de “pequenas melhorias” do **C++** com relação ao **C**.

Contudo, as principais diferenças, que devem implicar em **estilos de programação** diferentes, são justamente as “**grandes melhorias**” do **C++**: como veremos adiante, essas são os recursos voltados para **orientação a objetos** e para **programação genérica**.

E, conseqüentemente, esse será o **assunto mais importante deste curso**.

*Por enquanto, vamos apenas sintetizá-las:*



**C++** é maior e melhor que **C** principalmente porque incorpora a noção de **classes (orientação a objetos) e templates** ou “*gabaritos*” (**programação genérica**).

- **Esses** recursos ampliam consideravelmente a força do **C** e devem conduzir-nos a um **novo estilo de programação**, seja no que diz respeito ao uso da **memória**, seja no que diz respeito à **lógica** de programação, como também ao **reaproveitamento de código já escrito**.
- Esse assunto só será visto mais a frente. Por enquanto, basta saber que **Orientação a objetos** e **Programação Genérica** são técnicas de programação que permitem a escrita de um código mais **seguro**, confiável e fácil de manter, bem como possibilitam que um código escrito para situações genéricas e previsíveis possa ser **reaproveitado** mais

facilmente na criação de camadas de código mais específicas, sem que esse código novo interfira no código já escrito.

## 1.3 • Biblioteca de C++ e bibliotecas auxiliares

Bibliotecas oferecem recursos já desenvolvidos, aplicáveis a muitas situações e por isso permitem elevar a **produtividade** do programador, que não precisará reinventar a roda, criando código para uma série de áreas de desenvolvimento previsíveis e comuns. Algumas vezes precisamos desenvolver nossas **próprias** (e novas) bibliotecas, quando nos deparamos com situações para as quais não existem soluções prontas, ou então para resolver problemas específicos que se manifestem repetidamente em várias partes de um determinado projeto, ou de vários projetos.



Mas, antes de fazer isso, é importante analisar **se já não existe** alguma **biblioteca pronta** que resolva o problema.

### 1.3.1 • Biblioteca padrão de C++

C++ herda a biblioteca de funções de C. Isso significa que para uma grande variedade de situações previsíveis, já contamos com funções prontas para uso.

**Mais importante:** C++ acrescenta muitos novos recursos de biblioteca que, além de permitir a substituição (com vantagem) das funções herdadas (na maioria das situações), também amplia enormemente as funcionalidades oferecidas pela biblioteca herdada. Em especial, a *Standard Template Library*, ou **STL** (que veremos no devido tempo), threads, objetos de sincronização de threads, expressões regulares, entre outras.

### 1.3.2 • Bibliotecas adicionais

Além disso, atualmente, C++ conta com o apoio de uma infinidade de bibliotecas *open source* desenvolvidas por terceiros, das quais **vale destacar**:

- **boost**: é praticamente uma biblioteca *oficiosa* de C++; pois ela é desenvolvida por diversos dos integrantes do comitê de padronização de C++, e é, na prática, considerada a “porta de entrada” para que um novo recurso venha, no futuro, a fazer parte da biblioteca padrão de C++;
  - isso significa que ela oferece, hoje, recursos ainda não disponíveis na biblioteca oficial; ela não é inteiramente multiplataforma (como a biblioteca padrão); contudo, abrange as principais plataformas da atualidade (**Unix/Linux, Windows e Mac**).
  - veja: <http://www.boost.org>
- **Qt**: contem muitos dos recursos que já encontramos na **boost** (e até na biblioteca padrão), mas oferece recursos adicionais, como *interface* gráfica de usuário, suporte a serviços, bancos de dados, *Open GL, Web*, etc., além de oferecer suporte para certos ambientes embarcados;
  - do mesmo modo que a **boost**, não é inteiramente multiplataforma; mas, igualmente, abrange **Unix/Linux, Windows e Mac**; além disso, abrange também alguns ambientes *mobile* e embarcados: **Android, iOS, Windows RT, Embedded Linux, e alguns sistemas real-time como QNX**.
  - veja: <http://www.qt.io/>
- E há muitas outras bibliotecas, como **gtkmm, wxWidgets**, e, ainda, diversas outras voltadas para áreas específicas.

### 1.3.3 • Bibliotecas adicionais: indispensáveis

Nos últimos anos, o padrão da Linguagem C++ tem evoluído rapidamente, mas mesmo assim ainda não cobre todas as áreas vitais para se desenvolver alguns tipos de aplicações. Assim, precisaremos contar com **bibliotecas adicionais** que sejam **portáveis** ao menos para as **principais plataformas**: *Unix/Linux, Windows e Mac* - e, se possível, alguns sistemas embarcados. *Entre essas áreas de aplicação, destacamos:*

- **processos** e comunicação entre processos;
- **memória compartilhada** e memória mapeada;
- **network / sockets**;
- **acesso a banco de dados**;
- **programação gráfica**;
- **interface gráfica** de usuário.

**Alternativas:**

#### 0 Bibliotecas adicionais próprias:

- **Biblioteca padrão + [ boost + ] bibliotecas próprias [ + ... ]**

Supondo-se que exista o **conhecimento prévio** das APIs envolvidas pelas plataformas desejadas, precisaríamos desenvolver nossas próprias bibliotecas para cobrir todas ou algumas dessas áreas. **Inventar a roda...** Com isso teríamos os seguintes problemas:

- custo inicial de desenvolvimento;
- criar uma documentação aceitável;
- custo de manutenção, no qual deve estar incluído o treinamento de novos programadores que venham a ser agregados à equipe;
  - assim, o desejável é que o desenvolvimento de bibliotecas **próprias** esteja **restrito** às áreas de desenvolvimento, **onde não contamos com a cobertura de bibliotecas bem conhecidas**.

#### 1 Bibliotecas adicionais públicas; entre as alternativas disponíveis temos:

<ul style="list-style-type: none"> <li>■ <b>Biblioteca padrão + [ boost + ]</b></li> </ul>	<ul style="list-style-type: none"> <li>• <b>gtkmm (GTK+)</b> ;</li> <li>• <b>Qt</b> ;</li> <li>• <b>wxWidgets</b> ;</li> <li>• <b>outras.</b></li> </ul>	<b>[ + ... ]</b>
--	--	------------------

**Escolher nem sempre é fácil. Mas podemos ter alguns critérios:**

- **estabilidade e vitalidade** (presença no mercado e atualização de versões);
- **abrangência: plataformas e áreas de atuação** suportadas;
- **documentação**;
- **simplicidade** de uso;
- **licenciamento**

No caso das bibliotecas citadas acima, além da própria biblioteca oficial e da **boost**, considerando-se aqui **gtkmm**, **Qt** e **wxWidgets**, temos três boas opções. E, tudo pesado, escolhemos **Qt**. Com relação a **gtkmm** e **wxWidgets**, podemos dizer que **Qt** é:

- mais **completa** (tanto em plataformas suportadas como na amplitude das áreas de aplicação);

- melhor **documentada** e mais **simples** de usar;
- estabilidade e vitalidade comprovadas, pois a **atualização de versões** (e correções) é a mais frequente e sua **presença no mercado** é importante e crescente, pois hoje encontramos **Qt** em uma infinidade de aplicações, entre elas aplicações muito populares e marcantes como:
  - **KDE, Google Earth, Skype, simuladores de voo da NASA, etc.**
  - veja a relação completa em: [blog.qt.io/blog/category/qt-in-use/](http://blog.qt.io/blog/category/qt-in-use/).

Com relação ao licenciamento, até março de 2008 **Qt** perdia nesse critério, pois sua licença *open source* era restrita à GPL, ao passo que **gtkmm**, e **wxWidgets** podiam ser utilizadas sob a licença LGPL. Mas, a partir dessa data, com a versão 4.5 (e posteriores), **Qt também pode ser utilizada sob a licença LGPL**. Assim, não temos dúvidas em nossa escolha: **Qt**.

## 1.4 • Porque aprender C++

### 1.4.1 • Base sólida em programação

É importante aprender uma linguagem que suporta praticamente todas as técnicas de programação (e, em **C++11**, além de *orientação a objetos*, *programação genérica e rudimentos de programação funcional*, temos mais recursos para suporte à *programação funcional*). Isso propiciará uma **formação mais sólida para um bom programador**, pois ele perceberá que o mundo da **programação não se restringe a uma única técnica ou estilo**.

Essa base será útil, independentemente da linguagem que esse programador estiver usando em cada momento.

### 1.4.2 • C++ pode ser a linguagem principal

O que dissemos acima é valioso, mas não podemos esquecer que **C++ é útil em si mesma**: ou seja, para uma grande variedade de situações ela se apresenta como uma das melhores soluções, particularmente quando precisamos de **alta performance e portabilidade** entre diferentes plataformas.

Além disso, pela sua amplitude e clareza lógica, o aprendizado de C++ torna mais fácil e seguro o aprendizado de outros linguagens, porque serve de parâmetro. Não por acaso, outras linguagens famosas são descendentes de C++, ainda que não dispondo de todos os seus recursos.

### 1.4.3 • Usando outras linguagens

O fato de considerarmos **C++ como a linguagem mais completa já criada**, e até mesmo a adotarmos como **linguagem principal**, não deve servir para conclusões extremadas, do tipo: "só uso C++ e ponto final".

Linguagens de Programação são ferramentas. E há casos em que determinada linguagem pode ser mais simples e/ou eficiente para resolver um problema em particular.


Observe o exemplo do **Google**. Uma das qualidades dessa empresa é a sua versatilidade. O **Google** tem aplicações completamente feitas em C++ (como também tem aplicações em **Java**).



Por exemplo: o *Google Earth*, foi totalmente desenvolvido em C++, com o apoio da biblioteca Qt.

Já a sua aplicação mais tradicional, o *search engine* (a ferramenta de busca que fez a fama do *Google*), foi desenvolvida com C, C++ e Python.

---

 Em um famoso artigo dos criadores do *Google* ("*The Anatomy of a Large-Scale Hypertextual Web Search Engine*", de Sergey Brin e Lawrence Page), podemos ler que:

**"A maior parte do Google está implementada em C ou C++ por eficiência".**

E nesse mesmo artigo é dito também que para determinadas atividades (*web-crawler*) foi utilizada a linguagem **Python**.


---

Veja esse artigo (vale a pena) em:

<http://infolab.stanford.edu/~backrub/google.html>

E é inteiramente possível aplicar essa “dobradinha” **em muitas outras situações**: usando-se C++ no “motor” (o processamento pesado) onde a *performance* é mais crítica, e Python na ponta.

---

 Aliás, **Python** combina muito bem com C++, e não por acaso a maioria da comunidade de programadores **C++** admira (quando não usa) **Python**.

---

Ou seja: linguagens de programação não são times de futebol, pelos quais torcemos. São ferramentas. E escolher a(s) ferramenta(s) adequada(s) é um dos passos para bem solucionar problemas.

## 1.5 • Livros e sites sobre C++

Há diversos livros sobre C++ que podemos considerar como muito bons.

Mas devemos tomar um cuidado especial ao escolher um livro sobre C++: saber se ele reflete o **padrão da linguagem**. Conforme comentado acima, o primeiro padrão só foi firmado entre 1995 e 1998. E o padrão da linguagem incorporou um conjunto de melhorias, dos quais os mais importantes são *templates* e a STL (*Standard Template Library*).

E ainda há em circulação livros que **não incorporaram essas melhorias**. Isso significa que eles refletem o C++ anterior ao padrão, o qual é oficialmente considerado obsoleto.

**Abaixo, indicamos alguns dentre aqueles que consideramos os melhores livros sobre C++:**

### 1.5.1 • Livros para o iniciante

- O **melhor livro para iniciantes** atualmente é ***Programming: Principles and Practice Using C++*** do criador original de C++, **Bjarne Stroustrup**.

Este livro foi produzido como material para uma cadeira de primeiro semestre de Ciências da Computação da Universidade do Texas. Ver:

<http://www.stroustrup.com/programming.html>.

**OBS:** A **segunda edição** desse livro já incorpora os padrões **C++11** e **C++14**. (editora Addison-Wesley; ISBN-10: 0321992784; ISBN-13: 978-0321992789).

## 1.5.2 • Livros para aprofundamento

- O **primeiro livro** que você deve ler para **aprofundar** seus conhecimentos sobre **C++** também é de autoria do criador original de **C++**, **Bjarne Stroustrup: *The C++ Programming Language***.

**OBS:** deve ser adquirida a **quarta edição**, que já cobre o padrão **C++11**.

(editora Addison-Wesley; ISBN-10: 0321563840; ISBN-13: 978-0321563842).



**Atenção:** esse é um livro fundamental, sendo considerado uma expressão e um complemento do documento de padronização de **C++**.

**Não está dirigido a iniciantes e sim a programadores com algum domínio da linguagem.**

- ***The Design and Evolution of C++***, também de **Bjarne Stroustrup**;  
um excelente complemento ao livro anterior, onde você poderá aprender porque certas coisas são do jeito que são em C++, bem como as formas mais eficientes de uso da linguagem;  
(editora Addison-Wesley; ISBN 0-201-54330-3).
- ***The C++ Standard Library: A Tutorial and Reference*** de **Nicolai M. Josuttis**;  
um livro completo sobre a **biblioteca padrão de C++**; permitirá um conhecimento profundo da biblioteca, e servirá permanentemente como um livro de referência e consulta, quando iniciamos o uso de um novo recurso da biblioteca;  
A **segunda edição** já incorpora o **C++11**.  
(editora Addison-Wesley; ISBN-10: 0321623215; ISBN-13: 978-0321623218).
- ***C++ templates: The Complete Guide*** de **David Vandevoorde, Nicolai M. Josuttis e Douglas Gregor**;  
uma abordagem completa sobre um importante recurso de **C++: templates**;  
A **segunda edição** já incorpora **C++11, 14 e 17**.  
(editora Addison-Wesley; ISBN-10: 0321714121; ISBN-13: 978-0321714121).
- ***Modern C++ design: Generic Programming and design patterns Applied*** de **Andrei Alexandrescu**;  
expande o conhecimento sobre **templates** (além do bom uso de **herança múltipla**), explorando técnicas de programação genérica, com a aplicação de alguns *design patterns* bem conhecidos.  
(editora Addison-Wesley; ISBN-10: 0201704315; ISBN-13: 978-0201704310).
- ***C++ template Metaprogramming: Concepts, Tools, and Techniques from boost and Beyond*** de **David Abrahams**;  
expande o conhecimento sobre **templates** e programação genérica, avançando no terreno da **meta-programação**;  
(editora Addison-Wesley; ISBN-10: 0321227255; ISBN-13: 978-0321227256).

## 1.5.3 • Livros adicionais(abordagens e técnicas)

- ***C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*** de **Herb Sutter e Andrei Alexandrescu**;  
(editora Addison-Wesley; ISBN-10: 0321113586, ISBN-13: 978-0321113580).
- ***Effective C++: 55 Specific Ways to Improve Your Programs and designs (3rd Edition)*** de **Scott Meyers**;  
(editora Addison-Wesley; ISBN-10: 0321334876, ISBN-13: 978-0321334879).
- ***Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*** de **Scott Meyers**;  
(editora Addison-Wesley; ISBN-10: 0201749629, ISBN-13: 978-0201749625).

- ***Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*** de **Herb Sutter**;  
(editora Addison-Wesley; ISBN-10: 0201615622, ISBN-13: 978-0201615623).
- ***More Exceptional C++: 40 New Engineering Puzzles, Programming Problems, and Solutions*** de **Herb Sutter**  
(editora Addison-Wesley; ISBN-10: 020170434X, ISBN-13: 978-0201704341).
- ***Exceptional C++ Style: 40 New Engineering Puzzles, Programming Problems, and Solutions*** de **Herb Sutter**;  
(editora Addison-Wesley; ISBN-10: 0201760428, ISBN-13: 978-0201760422).
- ***More Effective C++: 35 New Ways to Improve Your Programs and designs*** de **Scott Meyers**;  
(editora Addison-Wesley; ISBN-10: 020163371X, ISBN-13: 978-0201633719).


## 1.5.4 • Alguns sites úteis sobre C++

- **cplusplus.com:**  
<http://www.cplusplus.com/>
  - Este é um excelente *site* para consultas sobre as **bibliotecas C e C++** (na seção **Library Reference**); além disso, oferece tutoriais, *forums*, artigos e informações em geral sobre C++.
  - E, por falar em **bibliotecas e referências sobre C++**, consulte também este:
- <http://www.cppreference.com/>  
**Obs:** este segundo site (*cppreference*) inclui, de modo mais completo, além das bibliotecas, informações aprofundadas sobre os recursos da linguagem em si.
- **Bjarne Stroustrup's homepage:**  
<http://www.stroustrup.com/>  
 Página do **criador original de C++**, Bjarne Stroustrup. Artigos, entrevistas, FAQs, *links* e informações diversas sobre C++.
- **The C++ Standards Committee:**  
<http://www.open-std.org/Jtc1/sc22/wg21/>
  - É o site do **comitê de padronização de C++**. Aí você poderá encontrar as propostas e discussões voltadas para a evolução de C++, especialmente para os novos padrões da linguagem: **C++11**, **C++14** e **C++17**.
- <https://isocpp.org/>  
  - E, para acompanhar a evolução do processo de **padronização**, além de **referências** e material diverso sobre **C++**, visite também este (**melhor** para consulta).
- **ACCU - Association of C and C++ Users:**  
<http://accu.org/index.php>
  - Discussões, artigos, notícias, resenhas de livros, *mailing lists*, *links*, *blogs*. Além disso, você pode tornar-se membro da associação. A página de *links* da ACCU merece destaque (pois evita que tenhamos que fazer aqui uma lista muito extensa..):  
<http://accu.org/index.php/articles/weblinks/c22/>
- **C++ Users Journal (atualmente é uma seção da mais prestigiada revista para programadores, a Dr. Dobbs's):**  
<http://www.drdobbs.com/cpp/>
  - O "CUJ" é a legendaria revista sobre **C++**. Artigos excelentes, código fonte, etc. Agora movida para a "Dr. Dobbs's".
- <https://www.cprogramming.com/>  
 Outro site com bom material de referência sobre **C** e **C++**.
- Como **lembrete**, repetimos aqui os *links*, para duas bibliotecas, já mencionadas acima, que consideramos fundamentais para o programador **C++**:
  - **boost:** <http://www.boost.org/>
  - **Qt:** <http://www.qt.io/>

E há ainda **muitos outros links**. Mas a relação acima é um bom ponto de partida para começar – e depois, através desses, encontrar os demais.

## 1.6 • Linguagem, Compiladores e Ambientes

Iremos iniciar agora um curso de **Linguagem C++**. A primeira coisa que precisamos entender é a **diferença** entre a **linguagem** e as **ferramentas** necessárias para gerar aplicações usando essa linguagem.

 A Linguagem C++ é um **padrão aberto**.

Assim sendo, podemos escrever um **código fonte** em **C++** que poderá ser usado para gerar aplicações em qualquer plataforma de *hardware* e Sistema Operacional - de pequenos dispositivos embarcados até micro-computadores e computadores de grande porte, e também em diferentes sistemas operacionais como, por exemplo, *Windows*, *Unix/Linux*, *Mac*, *Symbian*, *PalmOS*, *etc*.



Para isso, precisaremos utilizar **ferramentas de programação**, como compiladores, ferramentas de *debug* e ambientes de desenvolvimento.

Se a Linguagem em si é um padrão aberto, já as **ferramentas** são produtos – e neste caso encontramos ferramentas **gratuitas e comerciais**.

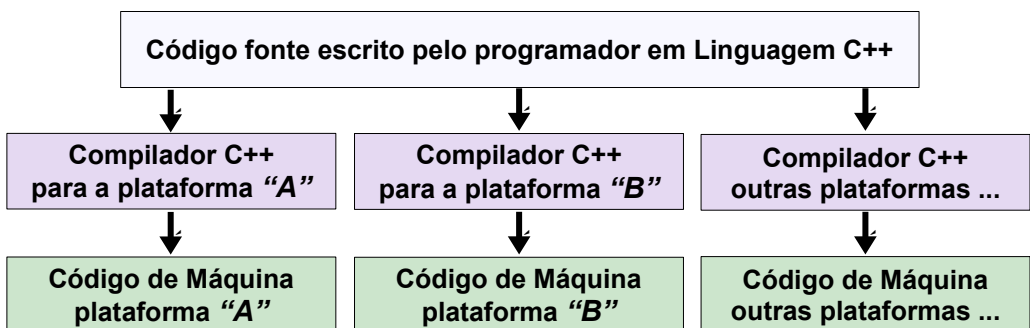
### 1.6.1 • Compiladores

Para desenvolver programas escreve-se o “*código fonte*” em uma determinada linguagem e em seguida utiliza-se uma ferramenta denominada “**compilador**”:

- o compilador (de cada linguagem) irá analisar a **sintaxe** empregada pelo programador em seu **código fonte**, de acordo com as **regras da linguagem**;
  - se a sintaxe estiver correta, ele irá **traduzir** esse código fonte para a linguagem da máquina, gerando instruções que possam ser “entendidas” e executadas por essa máquina, ou seja as “instruções de máquina” ou o “**código de máquina**”.
- Portanto, o programa estará convertido para as “**linguagens de máquina**”, específicas de cada plataforma, as quais, estando no nível da máquina, são classificadas como linguagens de “baixo nível”.



No caso de **C++**, usamos, em cada plataforma, o **respectivo compilador C++**, para que seja gerado o código de máquina para aquela plataforma.



Programar diretamente nas linguagens de máquina constitui um trabalho árduo, pois cada pequeno detalhe deve ser resolvido com uma instrução específica, a qual é identi-

ficada como um número binário, e o programador precisa lembrar os códigos numéricos das instruções (representados em hexadecimal). Além disso, sendo dependentes de plataforma, **linguagens de máquina não são portáteis**.

Para simplificar o uso das linguagens de máquina, foram criadas as linguagens “*assembly*”. Mas estas são constituídas apenas por expressões mnemônicas que representam os códigos binários das instruções de máquina, sendo assim também linguagens de baixo nível (ou *nível da máquina*). Pois cada mnemônico de *assembly* é traduzido diretamente para uma instrução de máquina.

E, como estão relacionadas diretamente às instruções de máquinas específicas, **também não são portáteis** entre diferentes plataformas.

Por isso mesmo as linguagens de programação mais utilizadas para a escrita de programas de computador são chamadas “**linguagens de médio e alto nível**”, em oposição às linguagens de “baixo nível”, por estarem mais próximas do “nível humano” e não da máquina, **simplificando** os detalhes. E, assim, muitas delas podem oferecer também uma única codificação para diversas plataformas.



Tais linguagens (e até mesmo *assembly*) não são compreendidas pela máquina e assim precisam ser **traduzidas** para a linguagem daquela máquina.

Este é justamente o papel dos **programas compiladores**.

Portanto, não basta conhecermos uma linguagem de programação. É necessário também dispor de **programas compiladores**. E há vários tipos de compiladores: alguns só conseguem gerar código de máquina para alguns sistemas operacionais (*Windows*, por exemplo); e há aqueles que dispõem de versões para diferentes sistemas operacionais. No caso da linguagem C++ dispomos de **diversos** programas compiladores. Há compiladores **gratuitos** e há compiladores **comerciais** cuja utilização exige a aquisição de licenças de uso.

## 1.6.2 • Ambientes de Desenvolvimento Integrado



Na maior parte dos casos, os *compiladores*:

- ou já são distribuídos **acompanhados de ambientes de desenvolvimento**;
- ou podem ser **utilizados em ambientes de desenvolvimento criados por terceiros**.

Um *ambiente de desenvolvimento* é uma ferramenta que visa integrar, em uma única interface de usuário, todas as ferramentas necessárias ao desenvolvimento de programas. *Como, por exemplo*:

- a. **editores de texto** voltados para a escrita de código fonte; esses editores, sendo especializados em programação, podem até identificar erros de sintaxe à medida em que o código é escrito, podendo ter diversos outros recursos auxiliares;
- b. como seria de se esperar, um ambiente desses deve conter recursos para **invocar** facilmente um **compilador** adequado para a linguagem em uso; alguns ambientes permitem até trabalhar com várias linguagens e compiladores.

- c. o mesmo para ferramentas de **detecção de erros (*debug*)**, integradas ao ambiente de forma a facilitar o acompanhamento da execução, inspeção de variáveis, etc.
- d. em diversos casos, os ambientes de desenvolvimento podem também gerar automaticamente alguns trechos de código para situações comuns e previsíveis.



E, do mesmo modo que ocorre com compiladores, temos à disposição diversos **ambientes de desenvolvimento, gratuitos e comerciais**.

### 1.6.3 • Compiladores comerciais e gratuitos

Há diversos compiladores para C++.

A escolha de um compilador pode estar baseada no fato de que ele seja **comercial ou gratuito**. Ou pelo fato de suportar totalmente (ou quase totalmente) o **padrão mais atual** da linguagem. Ou ainda, no fato de que determinado compilador gera um **código de máquina mais otimizado**, isto é, gera aplicações que executam mais rapidamente (melhor *performance*), como, por exemplo, o **GCC**, o compilador **Microsoft** e o compilador **Intel**. (este último só existe para máquinas com processadores da *Intel* e provavelmente, nessa plataforma, ainda é o que gera aplicações de melhor *performance*).

### 1.6.4 • Exemplos de compiladores

#### 1.6.4.1 • Gratuitos (alguns exemplos)

- **GCC** – em <http://www.gnu.org/software/gcc/index.html> – é um dos compiladores mais usados e tem versões para diferentes sistemas operacionais: **Unix/Linux, Windows, MacOS**(Macintosh), **Symbian, PalmOS**, etc.  
Há diversos **ambientes de desenvolvimento gratuitos** desenvolvidos para trabalhar em conjunto com o GCC.
- **Visual Studio Community** – em <http://www.visualstudio.com/pt-br/downloads>  
– é a edição gratuita da IDE da *Microsoft*, para estudantes, software livre e desenvolvedores individuais.
- **Apple XCode** – <http://developer.apple.com/technologies/tools/xcode.html> – é o ambiente de desenvolvimento da Apple, acompanhado de compiladores para **C, C++** e **Objective C**.
- **Digital Mars** – <http://www.digitalmars.com/> – edições gratuita e comercial.

#### 1.6.4.2 • Comerciais (alguns exemplos)

- **Microsoft**: – <http://www.visualstudio.com/pt-br/downloads/> – o compilador **C++** da *Microsoft* é vendido em conjunto com um ambiente de desenvolvimento: o “*Microsoft Visual C++*”, em diferentes edições comerciais (*professional, enterprise*, etc – a cada nova versão, confira as edições disponíveis no *link* indicado).
- **Intel**: – seguir os *links* para compiladores constantes na seguinte página: <http://www.intel.com/cd/software/products/asmo-na/eng/219763.htm>);  
**obs.:** o compilador da Intel dispõe de uma versão gratuita para *Linux*.
- **Comeau**: – <http://www.comeaucomputing.com/>.

### 1.6.4.3 • Outros exemplos

Diversos outros exemplos de compiladores gratuitos e comerciais (e ambientes de desenvolvimento) podem ser encontrados no seguinte *site*:

<http://www.stroustrup.com/compilers.html>

### 1.6.5 • Exemplos de ambientes de desenvolvimento

Um ambiente de desenvolvimento, como dissemos acima, permite **integrar** diversas **ferramentas** necessárias à criação de programas de computador.



O mais primitivo desses ambientes é o **arquivo *makefile***.

Em um arquivo “*makefile*” temos uma linguagem de *script* voltada para a compilação e *link-edição* de programas (que até pode ser usada para outros fins).

Quando precisamos compilar apenas um arquivo fonte (pequenos programas de teste, por exemplo), o uso do *makefile* não é tão importante. Mas, em outras situações, teremos uma grande vantagem, pois esse *script* permite relacionar **diversos arquivos fontes** além de outros recursos, como **bibliotecas**, que deverão ser considerados na geração dos binários. Desse modo evitaremos escrever, repetidamente, longas linhas de chamada ao compilador para cada nova recompilação.

Além disso ele permite identificar quais fontes precisam ser recompilados (apenas aqueles que foram alterados), reduzindo o tempo de compilação. Esse *script* é interpretado por programas que o entendem, como é o caso do utilitário “*make*”.

Contudo, o “*makefile*” permite apenas relacionar os arquivos necessários e as regras para gerar os binários. **Não está integrado** a editores de texto ou ferramentas de depuração de erros.

Já ambientes de desenvolvimento mais completos (ou **integrados**) tendem a aumentar a produtividade do programador, pois tornam mais rápida a execução de tarefas rotineiras e puramente burocráticas.

Acima, ao relacionar exemplos de compiladores, indicamos também que alguns deles já vêm acompanhados de um **ambiente**, como é o caso do “*Visual C++*”.

Além disso, há diversos outros ambientes, criados por terceiros - tanto comerciais como gratuitos.

**Nos links que relacionamos** na seção **1.6.4.3**, página **32**, acima, além de compiladores, serão encontradas também ambientes de desenvolvimento.

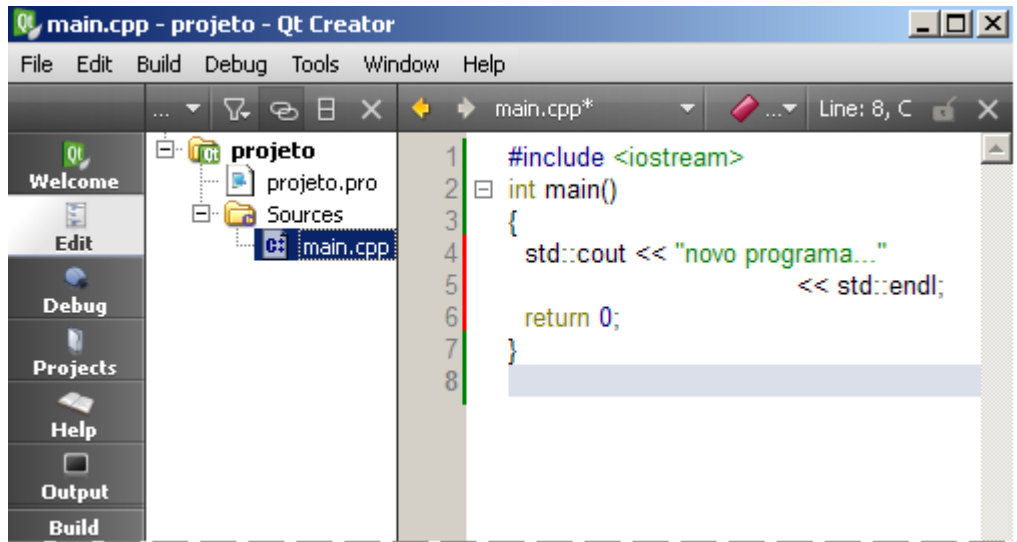
**Alguns exemplos de ambientes integrados de desenvolvimento (IDEs):**



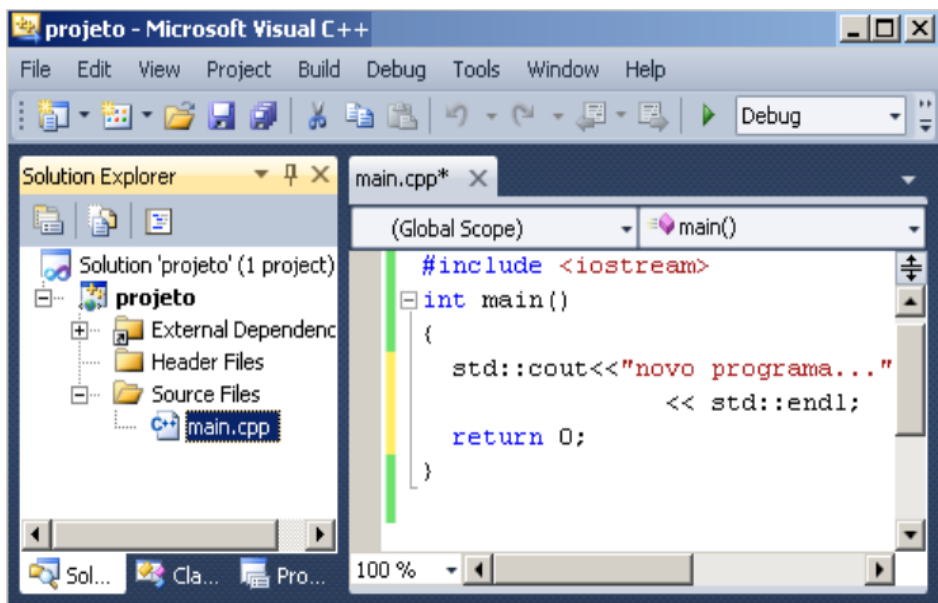
Imagens **meramente ilustrativas**; pois cada um deles pode ter uma **versão mais atual**; contudo, as funcionalidades essenciais são as mesmas.



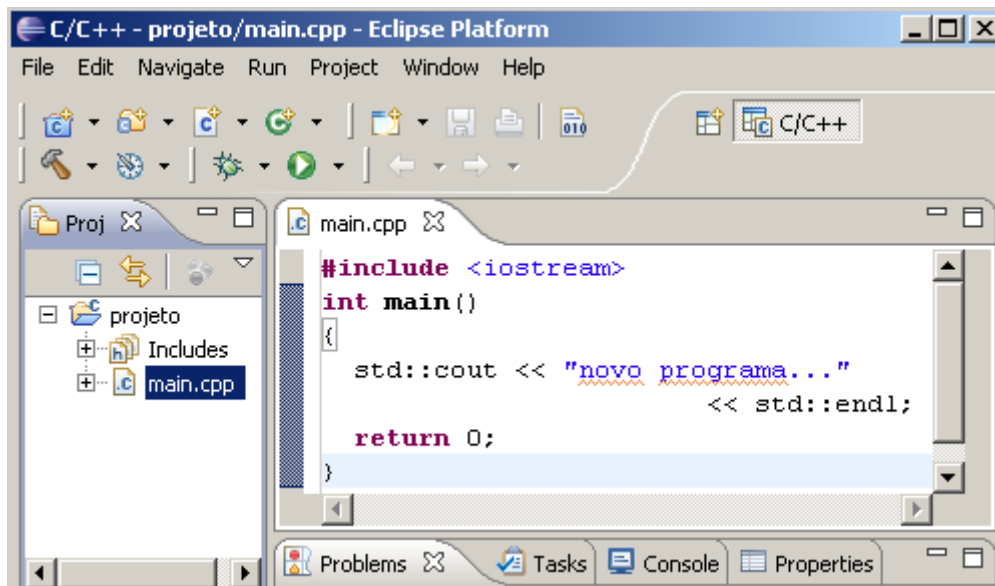
- **QtCreator:** (ver em <http://www.qt.io/>); gratuito, para *Windows, Unix/Linux e Mac*. Mais voltado para aplicações utilizando a biblioteca **Qt**. Contudo, pode ser usado para criar **qualquer aplicação C++** (mesmo sem usar Qt).



- **Microsoft Visual C++** (ver em <http://msdn.microsoft.com/visualc/>); já citado acima, com as edições comerciais e a gratuita (apenas *Windows*).



- **Eclipse** (ver em <http://www.eclipse.org/>); gratuito, para *Windows, Unix/Linux* e *Mac*.



- **KDevelop** (ver em <http://www.kdevelop.org/>); gratuito, para *Unix/Linux*.
- etc; veja os **links**, na seção **1.6.4.3**, página **32**, acima.

## 1.7 • O que veremos no curso

Neste curso veremos a **Linguagem C++** em seu padrão ISO (já com os principais recursos do *C++11, 14 e 17*).



Isto significa **que certos recursos dependentes de plataforma *não* são abordados neste curso.**

Mas serão abordados no curso **"Desenvolvimento multi-plataforma em C++ com Qt"** (*Windows, Unix/Linux, Mac, etc*).

Entre esses tópicos, dependentes de plataforma ou de bibliotecas específicas (não constantes do padrão da linguagem), e que **não** são vistos **neste** curso, destacamos os seguintes:

- **processos** e comunicação entre processos locais;
- comunicação entre **processos remotos** (*sockets, TCP/IP, portas seriais, etc*).
- **Memória mapeada em disco** (*memory map ou file mapping*)
- acesso a **banco de dados**;
- **programação gráfica** (*games, simuladores industriais ou para treinamento, etc*);
- **interface gráfica de usuário**;
- **aplicações web**.

## 1.8 • Conclusão

Procuramos nesta introdução dar uma ideia geral dos principais pontos de interesse e dos principais benefícios que você pode esperar em um curso de linguagem C++.

Para concluir, vamos sintetizar aqui o que consideramos mais importante no que foi acima exposto.

### 1.8.1 • Linguagem padronizada

Como C, C++ é também uma linguagem **padronizada**. Isto permitirá que  **muito** do seu código seja exatamente o mesmo nas suas aplicações para diferentes sistemas operacionais, como *Windows*, *Windows CE*, *Unix/Linux*, *MAC*, *Symbian*, *Palm OS*, etc.

Naturalmente há diferenças relativas aos sistemas operacionais (particularmente no que diz respeito à interface gráfica e processamento multitarefa, além dos demais tópicos que já citamos).

Mas, como vimos, há hoje bibliotecas já prontas, *open source*, que permitirão escrever um único código para todas essas plataformas, mesmo no que se refere aos tópicos específicos de plataforma. Além disso, há uma série de coisas em que as diferenças entre sistemas operacionais não interferem em absolutamente nada. Por exemplo, o código escrito para validar um CPF/CNPJ ou para transformar um valor em texto (valor por extenso) é o mesmo em *DOS*, *Windows*, *Linux*, etc...

E o que garante isso é justamente a **padronização da linguagem**.

### 1.8.2 • Linguagem multi-paradigma

Devido ao sucesso alcançado, C++ já tem hoje seus descendentes, como é o caso de **JAVA** e de **C# (C Sharp)** - embora estas sejam linguagens voltadas para plataformas específicas (plataformas *Java* e *.Net*, respectivamente).

Mas C++ é mais ampla que seus descendentes... Justamente porque é **híbrida** e suporta mais do que uma técnica de programação.

Boa parte do sucesso de C++ está na combinação de regras claras e expressivas com flexibilidade. Você não encontrará em C ou C++ muitas restrições arbitrárias; pelo contrário, na grande maioria das situações, as regras se combinam e se reaproveitam de **modo lógico**.

Uma vez entendidos os princípios básicos e aplicada sua lógica, torna-se fácil entender as implicações com reduzido espaço para as surpresas e exceções.

E é por isso que C alcançou tanto sucesso, tornando-se uma linguagem utilizada para praticamente todo e qualquer tipo de aplicação: de sistemas operacionais a editores de imagem; de aplicativos comerciais que exigem alta performance a sistemas de tempo real.

De tal modo, que todos ou quase todos os programas de propósito geral consagrados (esses mesmos que todos nós utilizamos em nosso dia a dia) foram escritos em C; e os mais recentes em C++.

E, em coerência com sua adesão à **flexibilidade**, a grande vantagem de C++ é suportar diversas técnicas de programação, como é o caso de *orientação a objetos* e *programação genérica*.

Além disso, também não retira nas mãos do programador os recursos necessários para enfrentar ambientes mais “hostis”.

Isto é, em dispositivos com menor capacidade de processamento ou memória, o programador poderá trabalhar mais próximo do nível da máquina (o popular “baixo nível”) de modo a compensar os problemas de *performance* e/ou escassez de memória.



Isto significa que **C++** é uma **linguagem de propósito geral**, voltada para a *programação de computadores* e não apenas para uma determinada plataforma, ou para um determinado tipo de aplicação.

Por isso, suporta múltiplos paradigmas de programação, sendo conhecida como uma linguagem **“multi-paradigma”**.

C++ foi concebida, deliberadamente, para admitir qualquer estilo de programação.

Podemos utilizar exclusivamente uma determinada técnica (como *orientação a objetos*) ou não.

Pois a linguagem não faz restrições ao programador. É ele quem decide o que vai usar, em função das necessidades de cada projeto específico.

Portanto, enquanto **JAVA**, por exemplo, é uma linguagem puramente orientada a objeto, já C++, no que diz respeito às técnicas de programação, não é uma linguagem pura e sim híbrida, isto é, uma linguagem que suporta **múltiplos paradigmas** de programação - ou **“multi-paradigma”**, como é costume dizer.



Dito de outro modo, **C++ não protege você de você mesmo...** (embora exija que você assuma explicitamente os riscos que pretende correr).

C++ oferece ao programador todos os recursos para que ele possa enfrentar qualquer situação possível em programação de computadores.



Inclusive, recursos para que **você possa se proteger de si mesmo**, se esse for o caso...

## 1.9 • Ferramentas usadas no curso

Compiladores:

- Microsoft: (apenas em *Windows*) – **não necessariamente estará disponível (mas você pode usar em seu notebook, caso deseje).**
- gcc: em *Windows* (se disponível) e *Linux*.

Inicialmente, no primeiro exercício, será mostrado como utilizar os compiladores diretamente na linha de comando. Para pequenos projetos, com um único arquivo fonte, isso é muito simples.

Mas isso muda de figura se passamos a ter mais arquivos fonte. Além disso, trabalhando na linha de comando, teremos que usar um editor de textos qualquer. E sempre que alteramos o fonte precisamos não esquecer de salvá-lo antes de uma nova compilação.

É por isso que ambientes de desenvolvimento integrados, embora, em princípio, não sejam indispensáveis, são uma ferramenta auxiliar de enorme importância para que o programador obtenha ganhos de produtividade.



Assim, após o primeiro exercício, os demais poderão ser desenvolvidos em um **ambiente integrado**.

**Ficará a critério de cada aluno escolher o ambiente** que considere mais prático ou mais completo.  
Inclusive, se preferir, poderá continuar usando a linha de comando...

Nesta apostila, mostraremos o uso de dois ambientes: **QtCreator** (pode ser usado em *Windows*, *Unix/Linux* e *Mac*) e **Visual C++** (apenas *Windows*).



Obs: o *Windows* e o *Visual C++* **não necessariamente estarão disponíveis**.  
**Mas se desejar você pode usar com o seu notebook.**

E, se você preferir, poderá usar o **Eclipse**.

O instrutor poderá usar qualquer um deles. E os alunos também farão suas escolhas: para os objetivos deste curso, **não faz qualquer diferença** qual o ambiente que você irá usar. E, sobretudo, você não é obrigado a usar o mesmo que o instrutor.








Do mesmo modo, você também pode escolher se **prefere trabalhar em Windows, Mac ou em Linux**: isso é **totalmente indiferente para este curso**

## 1.10 • Convenções usadas nesta apostila

### 1.10.1 • Versões de C++

- **C++03** é apenas uma "correção técnica" do **C++98**. Na realidade, o **C++03** visa acrescentar recomendações aos fabricantes de compiladores, mas não traz novidades para o programador.  
Por isso, nesta apostila, **não** iremos citar o **C++03** e **sim** apenas o **C++98** (fica subentendido que quando falamos em **C++98**, estamos nos referindo a "**C++98/03**").
- Além disso **não** iremos nos referir aos raros recursos de **C++98** que foram considerados "**descontinuados**" ou "desaprovados" ("**deprecated**") em **C++11** (e portanto marcados para remoção em futuras versões).
- Quando falamos em **C++11** estamos mencionando recursos válidos nessa versão e que continuam **válidos** nas versões seguintes.
- O mesmo quando nos referimos a **C++14** e a **C++17**.
- Caso algum recurso dessas versões venha a ser descontinuado no **C++20**, isso ainda é futuro. Hoje, isso não é uma realidade. Portanto, quando o **C++20** for aprovado e **liberado**, deveremos consultar a sua documentação e conferir se algum recurso anterior foi descontinuado. Por exemplo, isso pode ser feito no *site*:  
<https://www.cppreference.com>.

### 1.10.2 • Símbolos de atenção ou ênfase

-  Indica **ênfase**: algo que devemos **anotar** ou ter maior consideração (importante).
-  Indica **atenção**: algo que deve ser **evitado** ou usado com **cuidado** especial.
-  Indica **aprovação**: código **correto** e **adequado**.
-  Indica **desaprovação**: código **incorreto** ou **inadequado**.
-  Indica **erro grave**: seja **código errado** ou **erro de conceito**.

## 1.11 • Questões para revisão do Capítulo 1

Responda às questões abaixo, assinalando **todas** as respostas corretas (**uma ou mais**). Em seguida, compare suas respostas com as respostas localizadas no **Anexo B-1, página 425**. Caso não entenda, encaminhe as dúvidas ao instrutor.

### Cap.1 - 1. Com relação à **padronização** das linguagens **C** e **C++**:

- a. ☐ **C** é uma linguagem padronizada.
- b. ☐ **C++** é uma linguagem padronizada.
- c. ☐ **C não é** uma linguagem padronizada.
- d. ☐ **C++ não é** uma linguagem padronizada.
- e. ☐ A linguagem **C** já foi completamente padronizada, mas a padronização de **C++** ainda está em sua **fase inicial**.
- f. ☐ O padrão da linguagem C++ foi publicado em 1998, com uma atualização mínima em 2003. Isso é definitivo. **Nunca mais** haverá um novo padrão de C++.

### Cap.1 - 2. Assinale as afirmações verdadeiras sobre **C++**:

- a. ☐ **C++** introduziu apenas **pequenas** melhorias com relação ao **C**.
- b. ☐ **C++** é baseada **apenas** em **C**.
- c. ☐ **C++** é baseada em **C**, mas também herdou recursos decisivos de outras linguagens, o que torna **C++** muito diferente de **C**, em termos de técnicas de programação suportadas diretamente (ou nativamente).
- d. ☐ Além do **C**, outra linguagem que também serviu de inspiração para a criação do **C++** foi **JAVA**.
- e. ☐ **C++** suporta **apenas** uma **única** técnica de programação conhecida como “orientação a objetos”.
- f. ☐ **C++** é uma linguagem multi-paradigma, admitindo **múltiplas** técnicas de programação.

### Cap.1 - 3. Há uma série de áreas **dependentes de plataforma**, não cobertas pela biblioteca padrão de **C++**. Sendo assim, como atuar nessas áreas?

- a. ☐ Para resolver esse problema, o programador deve desenvolver **sempre** suas próprias bibliotecas, criando o código de infra-estrutura que cuide dos detalhes dessas plataformas, e sirva de base para a criação de aplicações que possam ser compiladas em qualquer uma delas.
- b. ☐ Já existem boas bibliotecas que resolvem a maior parte desses problemas, como, por exemplo, **boost** e **Qt**. Bastará escolher e usar a(s) biblioteca(s) mais adequada(s) para resolver esse tipo de problema, podendo assim criar aplicações que rodam em diversas plataformas.
- c. ☐ Os **compiladores** são responsáveis por resolver esse tipo de problema.
- d. ☐ Os **ambientes de desenvolvimento** são responsáveis por resolver esse tipo de problema.

## • Capítulo 2

### ▪ Iniciando: o básico em C++

Neste capítulo, veremos o básico para a **escrita de código** nessas linguagens.

Aqui estaremos quase sempre nos referindo a “**C** e **C++**”, pois o que será visto neste capítulo aplica-se, na maior parte, a essas duas linguagens.

**Exceto** no que diz respeito a recursos de biblioteca, onde sempre será dada **preferência à biblioteca C++**.

Além do básico sobre **C++**, veremos os passos necessários para **compilação** do código fonte, tanto em *Windows* como em *Unix/Linux*, bem como algumas dicas para a análise e correção de **erros** de compilação.

2.1 •	Tópicos de destaque neste capítulo.....	40
2.2 •	Exemplos preliminares.....	41
2.2.1 ▪	Um programa mínimo.....	41
2.2.2 ▪	Um programa um pouco maior.....	43
2.2.3 ▪	Blocos de código e chaves.....	45
2.2.4 ▪	Gramática básica.....	46
2.3 •	Compilando e executando.....	47
2.3.1 ▪	Exemplo no <i>Windows</i> .....	47
2.3.2 ▪	Exemplo em <i>Unix/Linux</i> .....	51
2.4 •	Analisando e corrigindo erros de compilação.....	54
2.5 •	Usando Ambientes de Desenvolvimento.....	56
2.5.1 ▪	Qt Creator.....	57
2.5.1.1 ▪	Criando um projeto.....	57
2.5.1.2 ▪	Adicionando arquivos ao projeto.....	61
2.5.1.3 ▪	Compilar e executar.....	62
2.5.2 ▪	Visual C++.....	63
2.5.2.1 ▪	Criando um projeto.....	63
2.5.2.2 ▪	Adicionando arquivos ao projeto.....	65
2.5.2.3 ▪	Compilar e executar.....	66
2.6 •	Conclusões sobre o código usado.....	67
2.7 •	Revisão do Capítulo 2.....	68
2.7.1 ▪	Exercício.....	68
2.7.2 ▪	Questões para revisão.....	69

## 2.1 • Tópicos de destaque neste capítulo

### ➤ Usando alguns dos recursos básicos de programação em C e C++.

- ◆ Funções.
- ◆ Chamadas de função.
- ◆ A função **main**: o ponto de entrada de uma aplicação (onde sua execução é iniciada).
- ◆ Linhas de instrução e operações.
- ◆ Tomadas de decisão com if (...) ... else ...
- ◆ Recursos de impressão da biblioteca C++.
- ◆ Sintaxe para o uso de cadeias de caracteres.
- ◆ Gramática básica para a escrita de nomes de símbolos, funções, linhas de instrução.

### ➤ Compilação.

- ◆ Utilizar os compiladores *Microsoft* e *gcc* para gerar os executáveis, tanto no *Windows* como no *Linux*.
- ◆ Executar os programas já compilados.
- ◆ Analisar e corrigir os erros de sintaxe do código fonte apontados pelo compilador.



## 2.2 • Exemplos preliminares

Antes de entrar em detalhes, vejamos dois **exemplos** de escrita de código válida em C e C++ (contudo, usaremos um recurso de **impressão que só é válido em C++**).

### 2.2.1 • Um programa mínimo

```
#include <iostream>
int main()
{
    std::cout << "imprime uma linha... e salta uma linha\n" ;
    return 0 ;
}
```

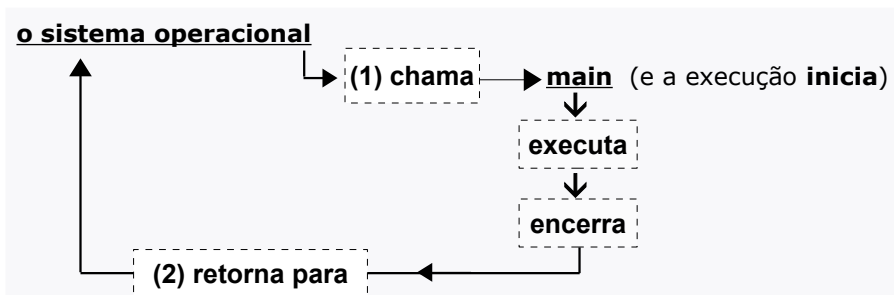
Como **resultado** da execução, será impresso no monitor de vídeo:

imprime uma linha... e salta uma linha  
[ Linha em branco, ou nada. Depende da plataforma. ]  
> [ próxima posição do cursor no monitor de vídeo ]

**Analisando cada elemento do código acima:**

- Acima temos uma **função**. Uma função é uma **sub-rotina** (que executa uma tarefa específica) e deve ter um **nome**, seguido por **parênteses**.

✎ E a função acima é denominada **main**. Esse é um **nome especial** e tem um significado: é nessa função que o programa **inicia sua execução**:



- Logo abaixo, no segundo exemplo, veremos o que faz o **"int"** antes do nome da função [ **"int main( )"** ] e porque ela termina com **"return 0"**.

- Uma função **agrupa instruções**. Esse grupo de instruções deve estar entre **chaves de início e fim**: { ... }

```
int funcao_qualquer ( )
{
    // início da função "funcao_qualquer"
    ... instruções ... ;
}
// fim da função "funcao_qualquer"
```

- Cada uma das **instruções** dever ser **concluída** com um **;** (**ponto e vírgula**):

```
std::cout << "imprime" ; // ponto e vírgula
```

- Por enquanto, não se preocupe com o símbolo `"std::"`, que aparece acima como um **prefixo** para o símbolo `"cout"`. Pense nesse prefixo como um nome de **"família"**, ou seja: **"cout" da família "std"**.  
Do mesmo modo, por exemplo, poderíamos fazer: **"Miranda::Basilio"**, ou seja: **"Basilio" da família "Miranda"**:

Família	::	Nome		Família	::	Nome
Miranda	::	Basilio		std	::	cout

- Mais a frente veremos o que significa exatamente esse prefixo (seção **7.5.10**, página **253**).
- O símbolo `"cout"` (abreviatura de **"character output"**) permite a **impressão** dos elementos que apareçam **após o operador "<<"**.
  - A expressão entre aspas após o **"operador <<"** (**"imprime uma linha... e salta uma linha\n"**) é uma cadeia de caracteres (ou **string**). Tudo o que o programa faz é justamente imprimir essa cadeia de caracteres.
    - Existem dois **caracteres especiais** no final dessa cadeia: `"\n"`. Uma `"\n"`, em uma cadeia de caracteres, representa o início de uma sequência **escape**.  
O que significa que o caractere que vier após a `"\n"` é um caractere de **controle**, interpretado de determinado modo.  
Nesse caso, um **"n" após uma "\n"**, provoca uma **quebra de linha** (**"newline"**). Logo, esse `"\n"` na verdade é convertido em um único caractere (o caractere de quebra de linha).

saída	← vai para	← cadeia de caracteres	← quebra de linha
<code>std::cout</code>	<code>&lt;&lt;</code>	<b>"imprime uma linha... e salta uma linha"</b>	<code>"\n"</code>

- Se quiser saber mais sobre "sequências escape" veja o **"guia de consulta rápida"**, no anexo **6**, página **486**.
- Na primeira linha desse programa mínimo temos:

**#include iostream**

- A diretiva **"#include"** avisa ao compilador que, caso ele não saiba o que significa um determinado símbolo (como é o caso do `"cout"`), ele deve procurar pela sua declaração em um determinado **arquivo**.  
Neste exemplo, é o arquivo **"iostream"**, no qual está declarado o que é e como pode ser usado o símbolo `"cout"`:

arquivo <b>iostream</b> - (requisitado pelo <b>"#include iostream"</b> )	
esse arquivo <b>declara "cout"</b> - ou seja: diz <b>o que é "cout"</b> e <b>como deve ser usado</b>	faz o mesmo para outros símbolos

- Até aqui descrevemos apenas o essencial sobre o nosso "programa mínimo". Mas, nos próximos dois capítulos tudo isso será melhor detalhado

## 2.2.2 • Um programa um pouco maior

```
#include <iostream>
int Maximo( int x , int y )
{
    if ( x > y ) // se "x" é maior-que "y"
        return x ; // retorna um resultado
    else // do contrário
        return y ; // retorna outro resultado
}
// ① o programa inicia aqui: ⤵

int main()
{
    int a, b, c ;
    a = 10 ; // copia '10' para 'a'
    b = 20 ; // copia '20' para 'b'
    c = ③ Maximo ( a, b ) ; ②
    std::cout << "O maior valor entre " << a << " e " << b
               << " = " << c << "\n" ;
    return 0 ;
}
```

( 1 ) A função "main" inicia a execução do programa.

( 2 ) A função "main" chama a função "Maximo", que executa suas instruções e, ao final, volta ao ponto em que foi chamada em "main".

( 3 ) Finalmente, em "main", o resultado de "Maximo" é copiado para "c".

Como **resultado** da execução, será impresso no monitor de vídeo:

O maior valor entre 10 e 20 = 20  
 [ linha em branco, omitida em algumas plataformas ]  
 > [ próxima posição do cursor no monitor de vídeo ]

**Além dos elementos que já descrevemos no exemplo anterior, neste exemplo temos os seguintes acréscimos:**

- Aparece aqui uma segunda função, denominada "**Maximo**" (cuja tarefa é encontrar o **maior de dois valores inteiros**) e que é **chamada** pela função "**main**". Isso significa que, quando ocorre a chamada, o processamento é desviado para as instruções que estão dentro de "**Maximo**", e quando são concluídas, o processamento retorna para o ponto em que ela foi chamada, em "**main**".
- Na função **main**, na primeira linha vemos: **int a, b, c ;**
  - Isso significa que pedimos uma **reserva de memória** para três números inteiros (é **isso que faz o "int"**), os quais serão acessados através dos nomes "**a**", "**b**" e "**c**", respectivamente. Assim, podemos armazenar valores em cada uma dessas posições ("**a=10;**" "**b=20;**"). E, depois, poderemos ler esses valores.

```
int a, b, c ;
a = 10 ; // copia '10' para 'a'
b = 20 ; // copia '20' para 'b'
c = Maximo ( a, b ) ; // copia o resultado de "Maximo" para "c"
```

**Três áreas na memória para armazenar números inteiros (int)**

" a "	" b "	" c "
10	20	será preenchida com o <b>resultado</b> da função " <b>Maximo</b> "

- Na função "**Maximo**" também temos pedidos de reserva de memória para dois números inteiros: "**x**" e "**y**". Esses pedidos aparecem em uma **posição especial**: entre os **parênteses** logo após

o nome da função.

Isso significa que essas memórias serão usadas como **parâmetros** da função (ou seja: valores de entrada, que a função necessita para executar seu trabalho):

```
Maximo ( int x , int y ) ;
```

" <u>x</u> "	" <u>y</u> "
quando a função for <u>chamada</u> , 'x' receberá um valor.	idem

- Quando "main" chama "Maximo" ela passa dois valores (**argumentos**) para **satisfazer esses parâmetros**:

```
c = Maximo ( a , b ) ;
```

- Isso significa que os valores de "a" e "b" serão lidos e **copiados** para "x" e "y", respectivamente.

```
c = Maximo (  a ,  b ) ;  
              ↓      ↓  
Maximo ( int x , int y ) ;
```

- A função "Maximo" faz uma **avaliação** que conduz a uma **tomada de decisão**:

```
if ( x > y )
```

- Se "x for maior que y" será **retornada** uma cópia do valor de "x":


```
return x ;
```

- **do contrário** será **retornada** uma cópia do valor de "y":

```
else  
    return y ;
```

- Isso explica porque aparece o "int" antes dos nomes das funções "main" e "Maximo":

```
int Maximo ( )  
...  
int main ( )
```

 esse "int", nessa posição, indica que cada uma dessas funções **retorna** um valor **numérico inteiro**, que representa o seu **resultado formal**.

- Após executar o "return", a função retorna para o ponto em que foi chamada em **main**, entregando o seu **resultado** (valor de **retorno**). Nesse exemplo, o resultado (que é um "int") é **copiado para "c"** (que também é um "int"):

```
c = <- Maximo ( a , b ) ;
```

- Quando "Maximo" **encerra** sua execução (retornando), as memórias alocadas por ela ("x" e "y") são automaticamente **liberadas**.

- Antes de "**int main( )**" vemos: "**// o programa inicia aqui:**". Isso é um **comentário** que é **iniciado com o símbolo "****//****"** e encerrado na quebra de linha ao final da sua linha de texto.  
Comentários são desprezados pelo compilador, mas são úteis para o programador que um dia precisará **reler** esse código - para corrigir erros ou melhorá-lo.

Inicia	Comentário	Encerra o comentário
//	o programa inicia aqui	[ quebra de linha do texto ]
//	<b>copia '10' para 'a'</b>	[ idem ]
//	<b>copia '20' para 'b'</b>	[ idem ]
etc...		

### 2.2.3 • Blocos de código e chaves.

Uma função contém um bloco de código (uma ou mais instruções) iniciado e encerrado com chaves: **{ ... }**. Conforme já foi exposto no primeiro exemplo, em uma função, as chaves **nunca** podem ser omitidas.

Além disso, dentro de uma função, podemos ter blocos internos para agrupar instruções que executem uma parte da sua tarefa.

Por exemplo: no caso do **if** podemos ter uma ou mais instruções a executar, conforme a condição seja avaliada como verdadeira ou falsa. Nesse caso, as chaves irão agrupar as instruções a executar para cada uma dessas duas alternativas. Mas, se houver apenas uma instrução, as chaves poderão ser omitidas.

Várias instruções: usar chaves.	Uma única instrução: as chaves podem ser omitidas.
<pre>if (x &gt; y) {     // uma ou mais instruções } else {     // uma ou mais instruções }</pre>	<pre>if (x &gt; y)     // uma única instrução else     // uma única instrução</pre>

## 2.2.4 • Gramática básica

Nos exemplos de código acima, usamos nomes para identificar memórias (por exemplo, "x" e "y") e escrevemos linhas de instrução. Para fazer isso foi necessário seguir as determinações da linguagem que formam sua sintaxe e sua gramática.

Em conclusão, para escrever “**nomes**” (ou melhor, **identificadores**) de **variáveis**, **constantes**, **funções e símbolos em geral** e também “**linhas de instrução**”, devemos seguir as seguintes regras básicas (há outras que veremos depois):

- **Palavras reservadas:** como qualquer linguagem de programação, **C** e **C++** oferecem um idioma, isto é um conjunto de palavras que identificam instruções e operações.
  - Por exemplo: nos exemplos acima usamos estas palavras reservadas: **int**, **if**, **return**. Há uma tabela completa das palavras reservadas no “**guia de consulta rápida**” (**anexo C-7**, página **486**).
  - Uma palavra reservada só pode ser usada para a finalidade que lhe foi atribuída pela linguagem.
- **Nomes de símbolos em geral:**
  - Em **C** somente os **32** caracteres iniciais de um nome de símbolo são considerados pelo compilador; em **C++** **não** há limite.
  - Devem **começar** com letra ou “**\_**” (sublinhado). Os outros caracteres podem ser letras, números ou “**\_**”.
    - **Exemplos:** **a\_1**, **a1\_** ou **\_a1** são nomes válidos.  
Mas **não:** **1a\_** ou **1\_a**
    - Em **C++**, para não incorrerem em conflitos com nomes de símbolos usados pelos fabricantes de compiladores na implementação da biblioteca padrão, temos uma **recomendação geral:** não usar o “**\_**” no início de um nome e também não usar um duplo “**\_**” em qualquer posição.  
Exemplos: **\_a1**, ou **a\_\_1** devem ser **evitados**. Prefira **a1\_** ou **a\_1**.
  - **Minúsculas e maiúsculas** são tratadas de forma diferente (**case sensitive**).
  - **Não podem** ter o mesmo nome de uma **palavra reservada**.
  - **Não podem** ter o mesmo nome **de uma função** definida anteriormente.
  - **Não podem** ter o mesmo nome de uma variável já definida no mesmo **escopo**. Veremos o que é um **escopo** mais adiante, na seção **5.6.6**, página **153**. Por enquanto, basta entender que não podemos ter duas variáveis com o **mesmo nome** no bloco de instruções de uma função, exceto se uma das declarações ocorre em um **bloco interno** (o bloco de instruções de um **if**, por exemplo).
- **Funções e instruções:**
  - O identificador (nome) para uma **função**, além de obedecer às regras descritas acima, deve ser acompanhado de **parênteses**.
  - Qualquer **bloco** (inclusive uma **função**) é **iniciado e encerrado** com **chaves**, exceto em controles de fluxo (como o **if**), onde podemos omitir as chaves se houver uma única instrução associada ao controle.
  - Uma linha de instrução é sempre encerrada com **;** (**ponto e vírgula**).

## 2.3 • Compilando e executando

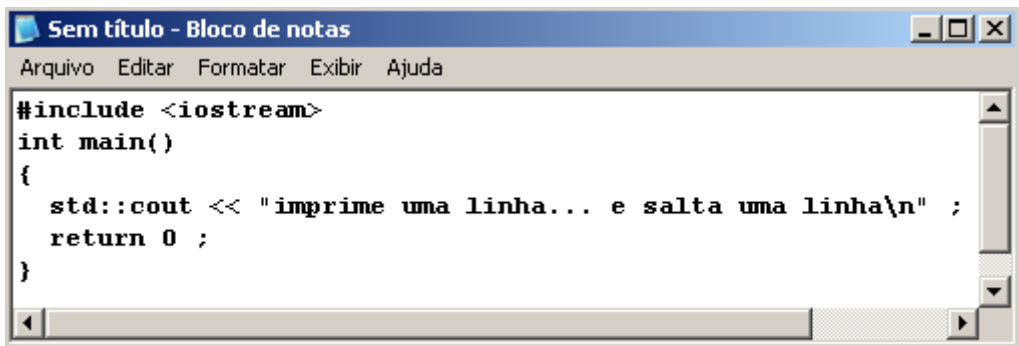
Vimos (na seção **1.6.1**, página **29**, acima) que o código escrito em linguagem C++ ou linguagem C é um código **fonte** que precisa ser **compilado**, isto é, traduzido para linguagem de máquina. Podemos colocar um compilador em execução diretamente **na linha de comando** ou **usando ambientes de desenvolvimento integrados**. Falaremos sobre ambientes ainda neste capítulo, mas vamos iniciar com a compilação diretamente na linha de comando, para melhor entender como funcionam os compiladores.

### Iniciando: compilação na linha de comando.

Para seguir esse caminho, em primeiro lugar precisamos escrever o código fonte, usando um editor de textos que não insira no texto do código caracteres especiais de formatação. Há vários editores desse tipo, como, por exemplo, o **notepad** ("bloco de notas") em *Windows*, ou os editores **vi**, **vim**, **gedit** e **kedit** em *Unix/Linux*.

### 2.3.1 • Exemplo no *Windows*

- a. Inicialmente vamos editar o "*programa mínimo*" (que está seção **2.2.1**, página **41**, acima) usando o "**bloco de notas**" do *Windows*.



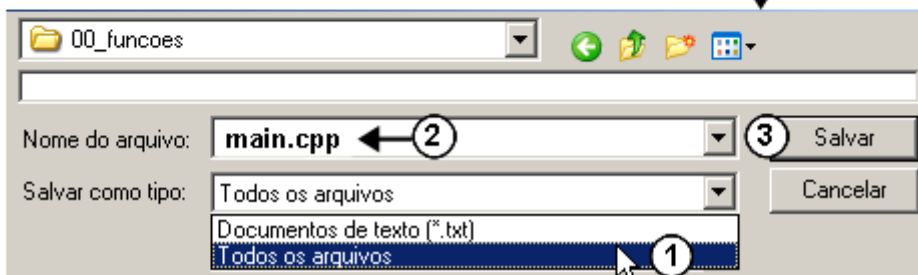
```
#include <iostream>
int main()
{
    std::cout << "imprime uma linha... e salta uma linha\\n" ;
    return 0 ;
}
```

- b. Agora é preciso **salvar o arquivo**, dando-lhe um nome.
- c. Para isso vamos inicialmente criar um **diretório** onde todos os projetos do curso sejam armazenados, para que depois sejam localizados facilmente.  
*Exemplo* (para *Windows*): **C:\cursoCPP** (crie o novo diretório "**cursoCPP**").
- d. Além disso, uma boa idéia é criar um **diretório específico** para cada um dos exercícios ou projetos, de modo a não misturar os seus arquivos fontes (pois alguns exercícios terão mais do que um arquivo fonte).  
Então, dentro desse diretório-base, **C:\cursoCPP**, vamos criar o sub-diretório para gravar os arquivos deste projeto.  
Por exemplo: **00\_funcoes**.  
Assim teremos o seguinte diretório de projeto: **C:\cursoCPP\00\_funcoes**.
- e. **Voltando ao "bloco de notas"**, já podemos **salvar** o primeiro arquivo fonte do "*programa mínimo*" no diretório de projeto criado acima:



Uma vez selecionado o diretório, agora:

- (1) selecione "Todos os arquivos",
- (2) informe o nome do arquivo ("main.cpp", por exemplo)
- (3) e pressione o botão "Salvar".



- f. Uma vez salvo o arquivo, resta agora **invocar o compilador**. Como já vimos, há vários compiladores para *Windows* que podemos usar. Neste exemplo usaremos **dois compiladores: microsoft e gcc**.

#### f.1. Microsoft.

Para **usar o compilador Microsoft na linha de comando** são necessários os seguintes passos:

(f.1.1) Instalar o compilador.

(f.1.2) Definir variáveis de ambiente no SO para acesso direto ao compilador.

(f.1.3) Chamar o compilador na linha de comando.

- f.1.1. Em primeiro lugar, você precisa **instalar** (versão **2010** ou **superior**):

- O *Visual Studio* da *Microsoft* em qualquer uma das edições comerciais.
- Ou a edição **Express** (gratuita) do *Visual C++*. Nas versões mais atuais essa edição é denominada como "*Community Edition*".

- f.1.2. Em segundo lugar, para **compilar na linha de comando** é preciso acrescentar algumas **variáveis de ambiente** nas propriedades do "*Meu Computador*". Mas isso pode ser feito de modo mais simples, definindo essas variáveis **diretamente no console** (ou terminal) onde você invocará o compilador na linha de comando.

- Para isso, podemos chamar um arquivo de lote que (geralmente) é instalado juntamente com o *Visual C++*: é o arquivo "**vsvars32.bat**".
- Em uma instalação típica, esse arquivo está localizado no diretório:

**C:\Arquivos de programas\Microsoft Visual Studio\<version>\Common7\Tools**

Onde "**<version>**" refere-se à versão do *Visual Studio* em uso.

Uma boa idéia é copiar esse arquivo para o diretório **C:\Windows\System32**.

Desse modo, ele estará sempre imediatamente visível para execução. Pois sempre que um **novo console** for aberto para compilação, esse arquivo em lote deverá ser executado **novamente**.

**copy vsvars32.bat C:\Windows\System32\**



- Agora, basta executar esse arquivo de lote, digitando na linha de comando (e concluindo com <enter>): **vsvars32**

```

C:\ Prompt de comando
C:\cursoCPP\00_funcoes>vsvars32
Setting environment for using Microsoft
C:\cursoCPP\00_funcoes>

```

- Se a resposta foi "Setting environment for using Microsoft Visual Studio" ou semelhante, então deve estar tudo pronto para que o compilador possa ser chamado na linha de comando.

### f.1.3. O terceiro passo é **chamar o compilador**.

Posicione-se no diretório **C:\cursoCPP\00\_funcoes** (e o arquivo **main.cpp**, que foi salvo acima, já deve estar aí). Então execute a seguinte linha:

**cl /EHsc /Fe"test" main.cpp**

```

C:\ Prompt de comando
C:\cursoCPP\00_funcoes>cl /EHsc /Fe"test" main.cpp
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00
Copyright (C) Microsoft Corporation. All rights reserved.
main.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation. All rights reserved.
/out:test.exe
main.obj
C:\cursoCPP\00_funcoes>

```

- Não se preocupe, por enquanto, com os detalhes. Na **linha de chamada ao compilador**, basta perceber que:
  - "cl" é o arquivo executável do **compilador microsoft**.
  - "main.cpp" é o **arquivo de código fonte** a ser **compilado**.
  - "test" é o **arquivo executável a ser gerado**.
  - Os outros detalhes serão vistos mais a frente.
- Se o compilador exibiu alguma **mensagem de erro**, será preciso analisar o código fonte. Logo abaixo, após os exemplos de uso dos compiladores, mostraremos alguns dos erros de compilação possíveis.

### f.1.4. Finalmente, o último passo é **executar** o programa gerado. Basta invocar o seu nome na linha de comando:

**test**

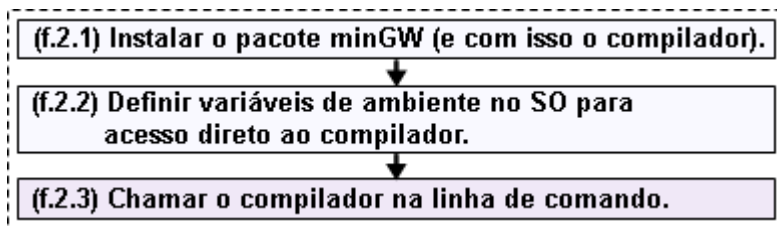
- A execução fez exatamente o que era **esperado**: foi impresso "imprime uma linha... e salta uma linha"; e em seguida houve uma quebra de linha, com uma linha em branco.

```

C:\ Prompt de comando
C:\cursoCPP\00_funcoes>test
imprime uma linha... e salta uma linha
C:\cursoCPP\00_funcoes>_

```

### f.2. **gcc**. Para usar o **compilador gcc em Windows**, você precisa **instalar** o pacote **minGW** (ou então o **cygwin**). Então, temos os seguintes passos:



**f.2.1.** Em primeiro lugar, você precisa **instalar**:

- O **minGW** (abreviatura de *Minimalist GNU for Windows*). Esse pacote oferece um porte de ferramentas GNU para *Windows*, como, por exemplo: **gcc**, **make**, **ar**, etc.

Oferece também os *headers* das bibliotecas de **C** e **C++**, bem como da *API Win32*, e, naturalmente, as próprias e respectivas bibliotecas.

Se você instalou algum ambiente de desenvolvimento como o **QtCreator** ou o **Eclipse**, o **minGW** já deverá estar instalado.

Do contrário, acesse <http://www.mingw.org>, para *download* e instalação.

**f.2.2.** Em segundo lugar, do mesmo modo que fizemos com o compilador *Microsoft*, para **compilar na linha de comando** é preciso acrescentar algumas **variáveis de ambiente** nas propriedades do "Meu Computador".

Mas, do mesmo modo, podemos simplesmente definir essas variáveis **diretamente no console** (ou terminal) onde você invocará o compilador na linha de comando.

- Caso você tenha instalado o ambiente **QtCreator**, **já existirá um arquivo de lote** que faz isso. Em uma instalação típica ele estará localizado aqui (o *path* poderá variar conforme a versão de **Qt**):

**C:\Qt\<version>\bin\qtenv.bat**

Você também pode copiar o arquivo **qtenv.bat** para o diretório

**C:\windows\system32**, para obter acesso direto e rápido.

- Caso não tenha instalado um ambiente (ou o ambiente não forneça um arquivo de lote desse tipo), podemos **criar um arquivo de lote** que deve conter as linhas exibidas abaixo. Use o "bloco de notas" para escrevê-lo.

Mas atenção para o *path* "**C:/minGW**": esse é o diretório padrão de instalação do pacote **minGW**. Modifique-o, caso tenha usado outro diretório.

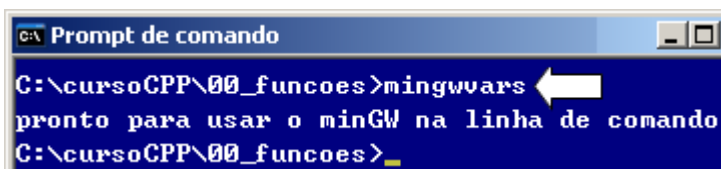
**@echo off**

**SET PATH=C:/minGW/bin;%path%**

**@echo pronto para usar o minGW na linha de comando**

- Agora salve o arquivo, nomeando-o. Por exemplo, use o nome **mingwvars.bat**. Copie o arquivo para **C:\windows\system32**.

- Execute esse arquivo de lote, digitando na linha de comando:  
**mingwvars.**



**f.2.3.** O terceiro passo é **chamar o compilador**.

Posicione-se no diretório **C:\cursoCPP\00\_funcoes** e execute a seguinte linha:

**g++ main.cpp -o test**

```

C:\cursoCPP\00_funcoes>g++ main.cpp -o test
C:\cursoCPP\00_funcoes>_

```

- Detalhes serão vistos mais a frente. Por enquanto, basta perceber os seguintes elementos na **linha de chamada ao compilador**:
  - "**g++**" é um atalho que chama o **compilador gcc** já devidamente posicionado para usar a **biblioteca** padrão de **C++** (e não apenas a biblioteca **C**). Se chamássemos o **gcc diretamente**, precisaríamos passar argumentos para incluir as bibliotecas de **C++**.
  - "**main.cpp**" é o **arquivo de código fonte** a ser **compilado**.
  - "**test**" é o **arquivo executável a ser gerado**. O "**-o**" indica que o nome que o segue é o nome do executável a gerar. E, como estamos em *Windows*, o arquivo gerado será "**test.exe**".
- Acima, o compilador não exibiu qualquer mensagem, o que significa que não localizou erros. Se, no seu teste, apareceram mensagens, provavelmente existem erros (ou avisos sobre procedimentos suspeitos). Daqui a pouco mostraremos alguns dos erros de compilação possíveis.

**f.2.4.** Finalmente, o último passo é **executar** o programa gerado, invocando o seu nome na linha de comando.

**test**

- E novamente ocorreu o **esperado**.

```

C:\cursoCPP\00_funcoes>test
imprime uma linha... e salta uma linha
C:\cursoCPP\00_funcoes>_

```

## 2.3.2 • Exemplo em *Unix/Linux*

**a.** Em *Unix/Linux* para editar o "*programa mínimo*", conforme já dissemos, podemos usar os editores **vi**, **vim**, **gedit**, **kedit** ou outros, dependendo da versão ou "sabor" do sistema operacional, bem como da distribuição instalada. Para simplificar, vejamos um exemplo com **gedit**.

```

#include <iostream>
int main()
{
    std::cout << "imprime uma linha... e salta uma linha\n" ;
    return 0;
}

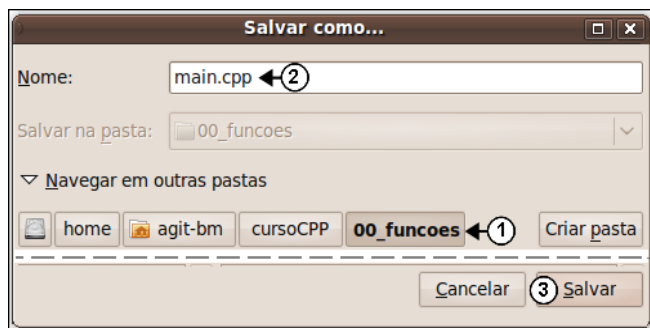
```

**b.** Crie um diretório de projetos ("**cursoCPP**", por exemplo), e, dentro dele, um diretório específico para este exemplo ("**00\_funcoes**", por exemplo), e finalmente salve o arquivo atribuindo-lhe um nome ("**main.cpp**", por exemplo).

Teremos então algo como:

**/home/user/cursoCPP/00\_funcoes/main.cpp**

onde "**user**" é o seu nome de usuário:



- c. Uma vez salvo o arquivo, resta agora **invocar o compilador gcc**, através do atalho **g++**. Para isso são necessários os seguintes passos:



- c.1. Instalação.** Normalmente, em uma instalação "desktop" típica do *Linux*, o **gcc** já deve estar instalado, mas é possível que o **g++** não esteja. Usando pacotes, ou os recursos disponíveis para instalação, é preciso garantir que os dois estejam instalados.

**Por exemplo**, no caso do **g++**, em uma distribuição *Debian*, bastaria fazer o seguinte:

**apt-get install g++**

```

agit-bm@ubuntu:~/cursoCPP/00_funcoes$ apt-get install g++ ←
Lendo listas de pacotes... Pronto
Construindo árvore de dependências
Lendo informação de estado... Pronto
g++ já é a versão mais nova.
  
```

Neste exemplo, o **g++** já estava instalado. Caso não estivesse, além do **g++** o "apt-get" já instalaria também suas **dependências**, como é o caso da **biblioteca C++**.

- c.2. Compilar.** Para chamar o **g++**, procedemos exatamente como na compilação em *Windows* mostrada acima:

**g++ main.cpp -o test**

```

agit-bm@ubuntu:~/cursoCPP/00_funcoes$ g++ main.cpp -o test ←
agit-bm@ubuntu:~/cursoCPP/00_funcoes$
  
```

- onde "**main.cpp**" é o **arquivo fonte a ser compilado** e "**test**" é o **nome completo do arquivo executável** (sem o ".exe").

- c.3.** Finalmente, resta **executar**, chamando o programa na linha de comando:

**./test**

```

agit-bm@ubuntu:~/cursoCPP/00_funcoes$ ./test ←
imprime uma linha... e salta uma linha
agit-bm@ubuntu:~/cursoCPP/00_funcoes$
  
```

- A execução fez exatamente o que era **esperado nessa plataforma**: foi impresso "*imprime uma linha... e salta uma linha*"; e em seguida houve uma quebra de linha, mas não houve a impressão de uma linha em branco (como no *Windows*).
- Isso **não significa** que o "**\n**" tenha perdido sua função. O que ocorre é que após a execução de um programa, nesta plataforma, o terminal não gera sua própria quebra de linha. Mas, se **eliminarmos o "\n"**, veremos que a impressão ficará confusa, juntando, **em uma mesma linha**, a linha impressa e a próxima exibição do "prompt".

```
std::cout << "imprime uma linha... e salta uma linha" ;
// sem o "\n"...
```

Teremos o seguinte resultado, **que não é o desejado**:

```
agit-bm@ubuntu:~/cursoCPP/00_funcoes$ ./test ←
imprime uma linha... e salta uma linhaagit-bm@ubuntu:~/
agit-bm@ubuntu:~/cursoCPP/00_funcoes$
```

## Compilando o segundo exemplo

- a. No arquivo **"main.cpp"**, acima da função **"main"**, **acrescente** a função **"Maximo"**, digitando o código que está no segundo exemplo (seção **2.2.2**, página **43**, acima) e que reproduzimos abaixo.

Em seguida **altere** a função **"main"**. O código completo deverá ficar assim:

Número da linha	Código (digite apenas o código - os números de linhas estão aí apenas para referenciar as linhas com erro nos exemplos seguintes).
1	#include <iostream>
2	int Maximo( int x , int y )
3	{
4	<b>if ( x &gt; y )</b> // se " <b>x</b> " é maior-que " <b>y</b> "
5	<b>return x ;</b> // retorna um resultado
6	<b>else</b> // do contrário
7	<b>return y ;</b> // retorna outro resultado
8	}
9	// o programa <b>inicia aqui</b> :
10	int main()
11	{
12	<b>int a, b, c ;</b>
13	<b>a = 10 ;</b>
14	<b>b = 20 ;</b>
15	<b>c = Maximo ( a, b ) ;</b>
16	<b>std::cout &lt;&lt; "O maior valor entre " &lt;&lt; a &lt;&lt; " e " &lt;&lt; b</b>
17	<b>&lt;&lt; " = " &lt;&lt; c &lt;&lt; "\n" ;</b>
18	<b>return 0 ;</b>
19	}

- b. **Salve o arquivo** mantendo o mesmo nome.

- c. **Compile** novamente, chamando na linha de comando:

```
cl /EHsc /Fe"test" main.cpp // compilador microsoft (Windows)
```

OU:

```
g++ main.cpp -o test // compilador gcc,
// seja em Windows ou em Unix/Linux
```



**O compilador emitiu mensagens de erro?** Falaremos de erros de compilação logo abaixo, na próxima seção.

- d. **Se não ocorreram erros, execute**, chamando na linha de comando:

```
test // Windows.
```

OU:

```
./test // Unix/Linux.
```

- **Resultado:** as linhas abaixo devem ter sido impressas no monitor de vídeo:


**O maior valor entre 10 e 20 = 20**

[ linha em branco, ou nada - dependendo da plataforma ]  
> [ próxima posição do cursor no monitor de vídeo ]

## 2.4 • Analisando e corrigindo erros de compilação

Temos abaixo alguns **erros muito comuns** quanto à sintaxe e regras da linguagem.

- a. Esquecemos de inserir um determinado **"#include"** necessário. **Exemplo:**

//  faltou o **"#include <iostream>"** (comente essa linha para testar)

```
int Maximo( int x , int y ) { ... }
```

```
int main() { ... std::cout ... } //  "cout" é declarada no arquivo iostream
```

- **compilar com o compilador microsoft (em *Windows*):**

```
cl /EHsc /Fe"test" main.cpp
```

E o compilador exibe a seguinte mensagem de erro:

```
main.cpp(16) : error C2653: 'std' : is not a class or namespace
main.cpp(16) : error C2065: 'cout' : undeclared identifier
// e aparecem outras mensagens decorrentes das anteriores.
```

Em **"main.cpp"**, na **linha 16**, o compilador **não reconheceu** dois símbolos: o **"std"** e o **"cout"** (**identificador não declarado**).


- **compilar com o gcc (em *Windows* ou *Unix/Linux*) :**

```
g++ main.cpp -o test // usando o gcc,
```

E o compilador exibe a seguinte mensagem de erro:

```
main.cpp:16: error: `cout' is not a member of `std'
```

Novamente, um erro em **main.cpp**, na **linha 16**. Neste caso a mensagem foi mais sintética. O compilador parece ter reconhecido o símbolo **"std"**, mas **não reconheceu "cout"** (e o código afirma que **"cout"** pertence à **"família std"**).

 Observar que, com mensagens diferentes, **os dois compiladores estão dizendo a mesma coisa:**

**"Nesse código, existem símbolos que eu, compilador, desconheço".**

E, quando o compilador diz isso, podemos ter duas situações:

- **faltou um #include** (esse é o caso deste exemplo);
- ou, em outras casos, um determinado símbolo foi escrito **de modo diferente de sua declaração inicial** (é o que veremos no próximo item).

 Para **localizar rapidamente qualquer erro** siga os seguintes passos:

- **Leia a mensagem de erro.** Ela dá uma indicação sobre o **tipo do erro**.
- Observe o **número da linha** em que o compilador afirma que ocorreu o erro.
  - Muitas vezes o erro **está nessa própria linha**.
  - Em outros casos, o erro é indicado em uma linha porque **faltou alguma coisa anteriormente**. E, analisando essa linha, veremos o que faltou acima dela.
    - Esse é o caso do nosso exemplo: é indicada a **linha 16**, porque **faltou** uma definição **prévia** do símbolo **"cout"**, que é usado **nessa linha**.

**b.** Ao escrever determinado símbolo, usamos nomenclatura **diferente da esperada**.

Isto é: usamos caracteres diferentes para referenciar um símbolo. Inadvertidamente, por exemplo, escrevemos "**count**" ao invés de "**cout**" ou então "**Cout**", com "**C**" **maiúsculo** (minúsculas e maiúsculas são consideradas diferentes). **Exemplo:**

```
#include <iostream> // OK, agora não esquecemos disso...
// ....
```

```
std::count ☹ << "O maior valor entre " ; // ...
```

OU:

```
std::Cout ☹ << "O maior valor entre " ; // ...
```

Usando "**count**", após a compilação teremos os **seguintes erros**:

■ **compilador microsoft:**

```
main.cpp(16) : error C2039: 'count' : is not a member of 'std'
main.cpp(16) : error C2065: 'count' : undeclared identifier
```

■ **compilador gcc:**

```
main.cpp:16: error: invalid operands of types <unknown type> ...
```

Os **dois** compiladores acusam um **erro** em **main.cpp**, na **linha 16**. E o erro está **exatamente nessa linha**.

As mensagens são diferentes, mas o sentido é o mesmo.

- O compilador **microsoft** diz: "'count' : is not a member of 'std'"; e também: "**count' : undeclared identifier**".
- O **gcc** diz: "**invalid operands of types <unknown type>**".

Dizer que um **identificador não foi declarado** ou que há uma operação envolvendo um operando de **tipo desconhecido** ("**unknown type**"), no fim das contas revela **o mesmo**: estamos usando um símbolo inesperado, que o compilador simplesmente não conhece.

E, usando "**Cout**", após a compilação teremos os **mesmos erros** apontados acima. Ou seja, a situação é a mesma: tanto "**count**" como "**Cout**" são **desconhecidos** pelo compilador.

**c.** Esquecemos do **;** (**ponto e vírgula**) para **encerrar** uma instrução:

```
#include <iostream>
```

```
// ....
```

```
std::cout << "O maior valor entre " << a << " e " << b
```

```
<< " = " << c << "\n" ☹
```

// Na linha acima **faltou o ponto e vírgula** para encerrar a instrução...

■ **compilador microsoft:**

```
main.cpp(18) : error C2143: syntax error : missing ';' before 'return'
```

■ **compilador gcc:**

```
main.cpp:18: error: expected `;' before "return"
```

Os **dois** compiladores acusam um **erro** em **main.cpp**, na **linha 18**. Mas o erro está **na linha acima (linha 17)**.

As mensagens são ligeiramente diferentes, mas o sentido é o mesmo:


**falta um ";" (ponto e vírgula) antes (before) da linha 18 ("return 0;").**

E realmente, **antes** de **"return 0; "**, ou seja, na **linha 17**, temos um **problema**:

```
<< " = " << c << "\n"
```

**Nessa linha**, falta um ponto e vírgula. Portanto, a mensagem é clara: falta alguma coisa **antes** da linha que o compilador estava avaliando.

d. Ao abrir ou fechar uma cadeia de caracteres esquecemos das **aspas duplas**:

```
#include <iostream>
// ...
std::cout << "O maior valor entre " << a << " e " << b
                                     << " = " << c << "\n"  ;
// Desta vez, na linha acima, não esquecemos do ponto e vírgula,
// mas esquecemos (ou eliminamos inadvertidamente)
// as aspas duplas de fechamento da cadeia de caracteres.
```

#### ■ compilador microsoft:

```
main.cpp(17) : error C2001: newline in constant
// e aparecem outras mensagens decorrentes da anterior.
```

#### ■ compilador gcc:

```
main.cpp:17: error: missing terminating " character
// e aparecem outras mensagens decorrentes da anterior.
```

Os **dois** compiladores acusam um **erro** em **main.cpp**, na **linha 17**. E o erro está **exatamente nessa linha**.

As mensagens, na aparência, são bem diferentes. Mas o sentido é o mesmo:

- o compilador **microsoft** diz que há uma **"newline"** (uma quebra de linha) no meio da "constante", que no caso é a cadeia de caracteres ("**\n**"), a qual não está terminada. Isso significa que antes que a cadeia fosse fechada com **aspas duplas**, apareceu uma quebra de linha inesperada.
- o compilador **gcc** diz (de modo mais claro) justamente que as aspas duplas estão ausentes na posição de fechamento da cadeia de caracteres: **missing terminating " character**. Traduzindo: **está faltando o caractere de terminação \_** (as aspas duplas).

## 2.5 • Usando Ambientes de Desenvolvimento

A compilação na linha de comando, conforme vimos acima, é bastante simples. Se tudo estiver instalado corretamente, bastará executar **uma dessas duas linhas**:

```
cl /EHsc /Fe"aplicacao" main.cpp // compilador microsoft - em Windows
```

```
g++ main.cpp -o aplicacao // compilador gcc - em Windows ou Unix/Linux
```

Contudo, essa simplicidade deixa de existir quando temos mais que um arquivo fonte a compilar, ou, ainda, bibliotecas adicionais que devem ser incorporadas à geração do executável.



Nesse caso, essas linhas tornam-se excessivamente longas, tornando improdutivo a tarefa de simplesmente chamar um compilador.

Além disso, **mesmo** com um **único** arquivo fonte, **sempre que alteramos** alguma coisa no código (usando o *notepad*, o *vim*, o *gedit*, ou qualquer outro editor), é preciso **lembrar de salvar as alterações**, ou do contrário a próxima chamada ao compilador irá compilar o arquivo existente (anterior à alteração, a qual, por enquanto, está apenas na memória do editor, e não no arquivo em disco).

Por isso o caminho mais conveniente e produtivo é usar **ambientes de desenvolvimento integrados**, onde o editor de textos e as chamadas ao compilador estão em um mesmo lugar.

Vejamos agora exemplos de uso de **dois ambientes**: **QtCreator** (este pode ser usado em *Windows*, *Unix/Linux* e *Mac*) e **Visual C++** (apenas *Windows*).

A partir daqui, **você poderá usar qualquer um deles, em qualquer SO suportado**. Ou se preferir, pode continuar usando a linha de comando...

O **Qt Creator** é o ambiente de nossa preferência pois pode ser usado em várias plataformas e não exige grandes quantidades de memória para sua utilização (é o mais "leve"), tendo todas as funcionalidades importantes que esperamos de um ambiente integrado.

Contudo, é possível que essa não seja a melhor escolha para você. Por exemplo, se o SO que você prefere é o **Windows** e se, além disso, você prefere usar o compilador **microsoft** (ao invés do **gcc**), então nesse caso o ambiente mais adequado é o **Visual C++**. Observar que **um ambiente não substitui o compilador**. Pelo contrário, uma de suas tarefas é justamente **chamar o compilador**, o que é feito quando pressionamos um determinado botão ou opção de *menu* destinados a fazer essa chamada.

O **QtCreator** e o **Eclipse**, tanto no *Unix/Linux* como no *Windows*, por *default*, chamam o compilador **gcc**. O **Visual C++**, naturalmente, chama o compilador **microsoft**.



Frisando: as imagens abaixo dos ambientes de desenvolvimento são **meramente ilustrativas**. Podem **variar** conforme a **versão** de cada ambiente. Mas as funções essenciais (edição, compilação e execução) permanecem as mesmas.

## 2.5.1 • Qt Creator

### 2.5.1.1 • Criando um projeto

Todos os ambientes trabalham com a ideia de "**projeto**". Um "projeto" é apenas um arquivo que reúne informações sobre o **tipo de aplicação** a ser gerado (se ela será um executável ou uma biblioteca, por exemplo), e mantém uma **lista de todos os arquivos de código fonte** necessários para gerar a aplicação, de modo que eles sejam incluídos na chamada ao compilador que é feita com recursos do próprio ambiente.

Cada ambiente, tem suas próprias regras sobre como o arquivo de projeto é escrito e nomeado.

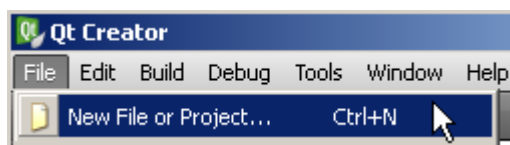
No Qt Creator, por exemplo, esse é um arquivo cujo nome tem a extensão **".pro"** e nesses arquivos são usadas algumas variáveis específicas do próprio ambiente, as quais visam a identificar as características daquele projeto.

De modo geral, não é preciso ter preocupações especiais com o arquivo de projeto. O que nos interessa são os arquivos de código fonte.

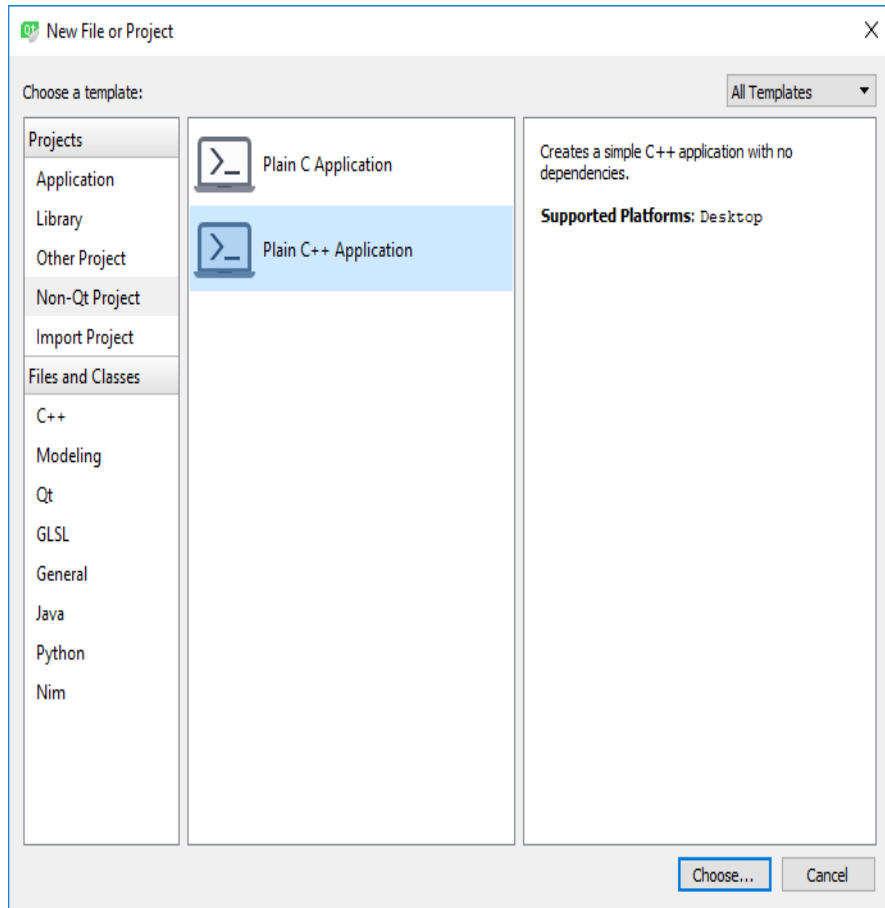
Depois, se for o caso, podemos criar projetos em outros ambientes e simplesmente anexar esses arquivos. Pois criar projetos, na grande maioria dos casos, é uma tarefa muito simples e que é levada a cabo com alguns movimentos e *cliques de mouse*.

Assim, o próximo passo é criar um projeto para poder trabalhar com o exemplo que vimos acima. Como já dissemos na introdução, o Qt Creator é especializado em trabalhar com a biblioteca **Qt**, mas pode ser usado para projetos que usam **apenas a linguagem C++** e a biblioteca padrão.

- a. Utilize a opção de *menu File*, e, em seguida, a opção **New File or Project**.



- b. Será aberta a janela abaixo. Nessa janela selecionamos o tipo de recurso que desejamos criar.. O que nos interessa aqui é um projeto sem o uso de Qt. Então selecionamos **Plain C++ Application**.



**c.** Agora, basta selecionar o tipo de projeto sem Qt que desejamos: **Plain C Application** ou **Plain C++ Application**.

No nosso caso, a opção que se aplica é **Plain C++ Application** (uma aplicação C++).

**(1)** Selecione o **tipo do projeto**.

**(2)** **Prossiga** para a próxima etapa.

**d.** Próximos passos:

**(1)** Informar o **nome do projeto**.

**(2)** O **diretório** em que será criado:

No *Windows* pode ser: **C:\cursoCPP**.

No *Unix/Linux* pode ser: **/home/<user>/cursoCPP**.

**(3)** **Prossiguir** para a próxima etapa.

Caso ainda não exista, será criado um diretório com o nome do projeto, abaixo do diretório base: **C:\cursoCPP\00\_funcoes** ou **/home/<user>/cursoCPP/00\_funcoes**.

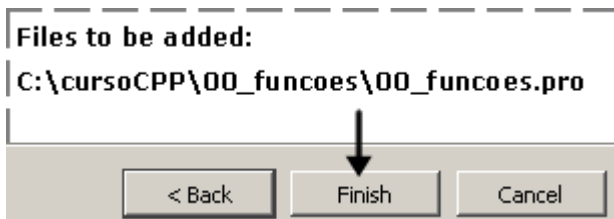
E, caso já exista, será usado o diretório existente.

A mensagem "*The project already exists*" indica na verdade que já existe um diretório **<...>/cursoCPP/00\_funcoes**. Neste caso, podemos despezá-la.

e. E atingimos a última janela.

Ela apenas informa que será criado um arquivo de projeto, com o nome **00\_funcoes.pro**, localizado no diretório do projeto.

Aqui, basta pressionar o botão **Finish**.



f. Agora, voltamos à área de trabalho.

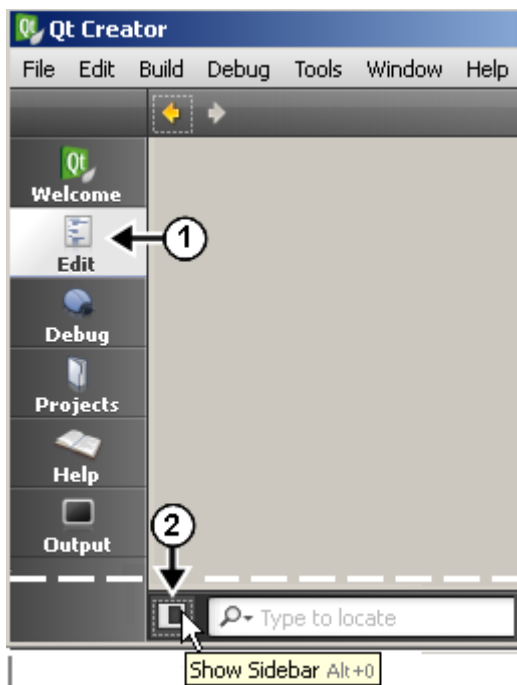
Para adicionar e editar arquivos, temos um **navegador de projetos**.

É bem possível que ele já esteja visível.

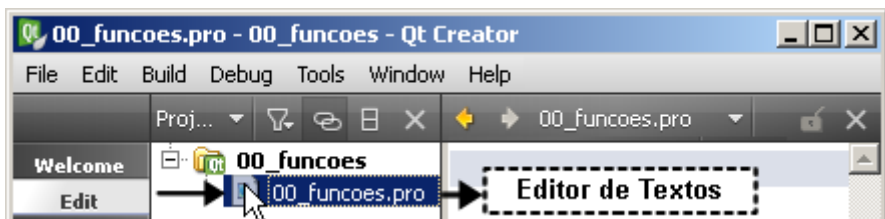
Mas, **caso não esteja**, basta:

(1) Garantir que o modo de trabalho seja **Edit**, na barra vertical à esquerda da área de trabalho.

(2) Pressionar o botão **Show Sidebar**, localizado na parte inferior da área de trabalho. Esse botão **exibe/oculta** o navegador de projetos (a *sidebar*).

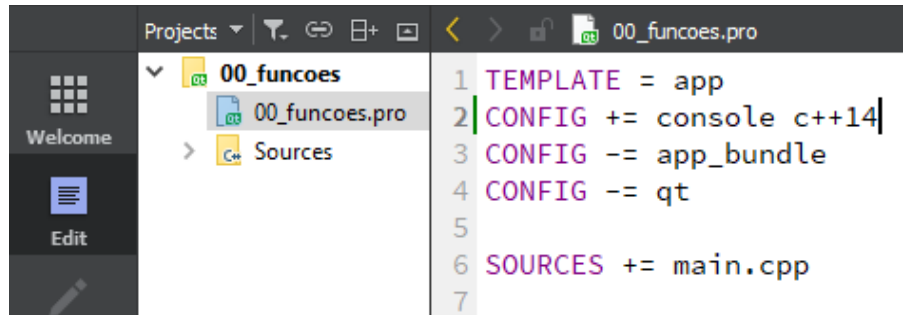


g. O navegador de projetos, à esquerda, após a barra vertical, exibe a lista de arquivos já anexados ao projeto. Neste caso, temos o "**arquivo de projeto**": **00\_funcoes.pro** e um módulo fonte C++ chamado **main.cpp**. Com um duplo clique sobre o arquivo de projeto, será aberto o editor de textos, para que este possa ser editado:



h. Quando o compilador usado é o GCC, precisamos modificar aqui a versão do C++ usada no programa, neste caso é a versão C++14.

**CONFIG += console C++14**



O projeto em si está pronto. Salve e feche o arquivo "**00\_funcoes.pro**", pois a partir daqui não haverá mais nada a fazer com ele. Simplesmente, esqueça que ele existe.

O QtCreator adicionou o módulo "**main.cpp**" ao projeto. Nesse caso, nós já temos um módulo fonte com o código que precisamos e não queremos este arquivo, então podemos removê-lo do projeto selecionando-o e depois pressionando a tecla **Delete**, selecionamos a opção "**Delete file permanently**" e então pressionamos **OK**.

### 2.5.1.2 • Adicionando arquivos ao projeto.

Para acrescentar arquivos temos duas opções:

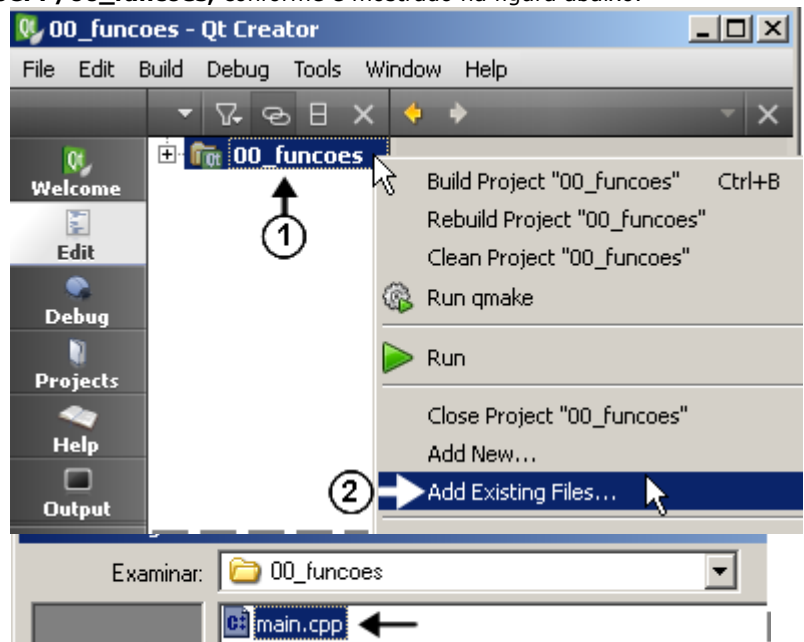
- **Add New:** acrescenta um novo arquivo.
- **Add Existing Files:** acrescenta arquivos já existentes.

No nosso caso queremos acrescentar um arquivo já existente: o arquivo **main.cpp** que está no diretório **<...>/cursoCPP/00\_funcoes**, conforme é mostrado na figura abaixo.

No navegador de projetos:

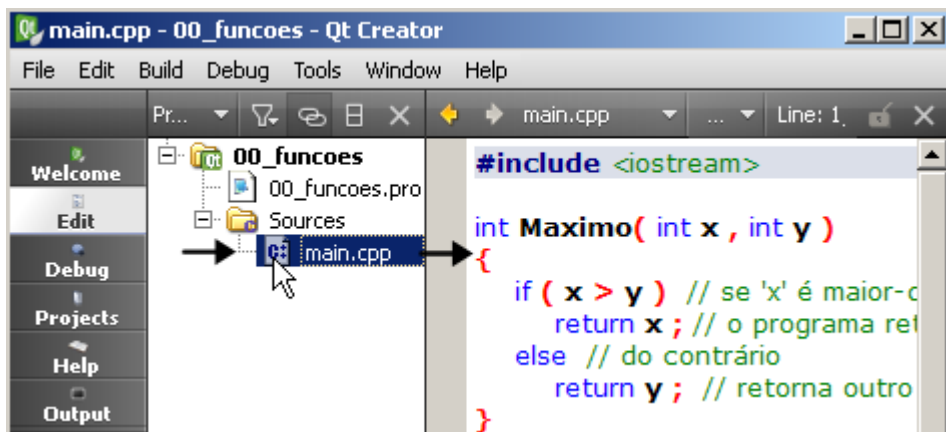
(1) Acione o botão direito do *mouse* sobre o nome do projeto.

(2) Selecione a opção do *menu* suspenso **Add Existing Files**.



Adicione o arquivo **<...>/cursoCPP/00\_funcoes/main.cpp**.

Após adicionar **main.cpp** ao projeto, vemos que o navegador de projetos já apresenta um nó para ele. Assim, o arquivo estará sempre diretamente disponível para edição:

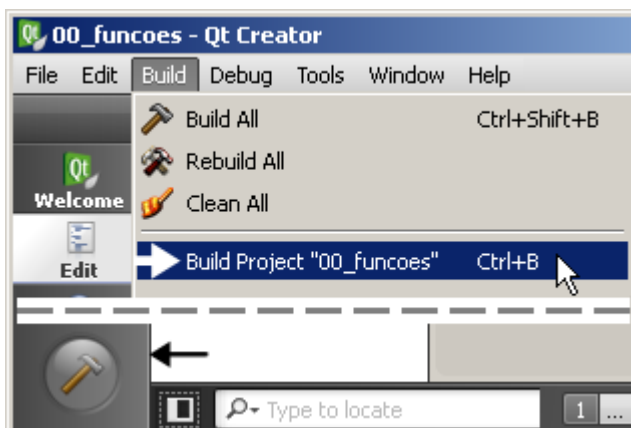


### 2.5.1.3 • Compilar e executar.

#### a. Compilar.

Para isso podemos usar a opção de **menu Build**.

Ou então o **botão de atalho** que está ao final da barra de ferramentas vertical, à esquerda da área de trabalho.



#### b. Finalmente: **executar** o projeto.

Para executar o programa, basta ir ao menu **Build** e selecionar a opção **Run**. Ou você pode pressionar o botão **Run** na barra vertical à esquerda da área de trabalho. Caso queira executar o programa na linha de comando, abra um terminal e vá para o diretório onde o executável foi gerado e simplesmente chame o executável na linha de comando.

No **Linux/Unix**, o **executável** estará no **próprio diretório do projeto**.

Já em **Windows** poderão ser criados dois diretórios, abaixo do diretório do projeto: **debug** e **release**. Nos anexos, veremos para que servem essas duas opções.

Por enquanto, se você não alterou nenhuma característica do projeto, basta saber que o **executável** estará no diretório **debug**:

```

C:\ Prompt de comando

Pasta de C:\cursoCPP\00_funcoes\debug
25/04/2010 16:52 <DIR> .
25/04/2010 16:52 <DIR> ..
25/04/2010 16:52 3.790.801 00_funcoes.exe

```

Então, basta chamar na **linha de comando**:

00\_funcoes // Windows.  
ou:

./00\_funcoes // Unix /  
Linux.

```

C:\ Prompt de comando

C:\cursoCPP\00_funcoes\debug>00_funcoes

0 maior valor entre 10 e 20 = 20

C:\cursoCPP\00_funcoes\debug>

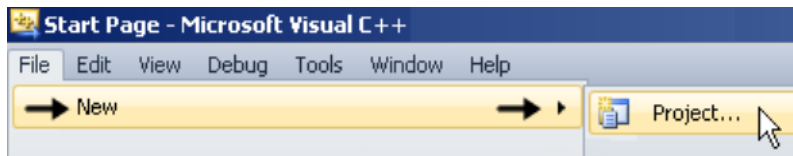
```

## 2.5.2 • Visual C++

### 2.5.2.1 • Criando um projeto

Os passos para a criação de um projeto no **Visual C++** são semelhantes aos que usamos no **QtCreator**.

a. Utilize a opção de *menu* **File**, e, em seguida, as opções **New** e **Project**.



b. Na próxima janela, para criar um projeto de aplicação executável usando apenas a Linguagem **C++** (sem nenhum recurso específico das plataformas *Windows* e *.NET*), deve ser escolhida a opção **Win32 Console Application**.

(1) Em *templates*, selecione **Win32**.

(2) Selecione o tipo do projeto: **Win32 Console Application**.

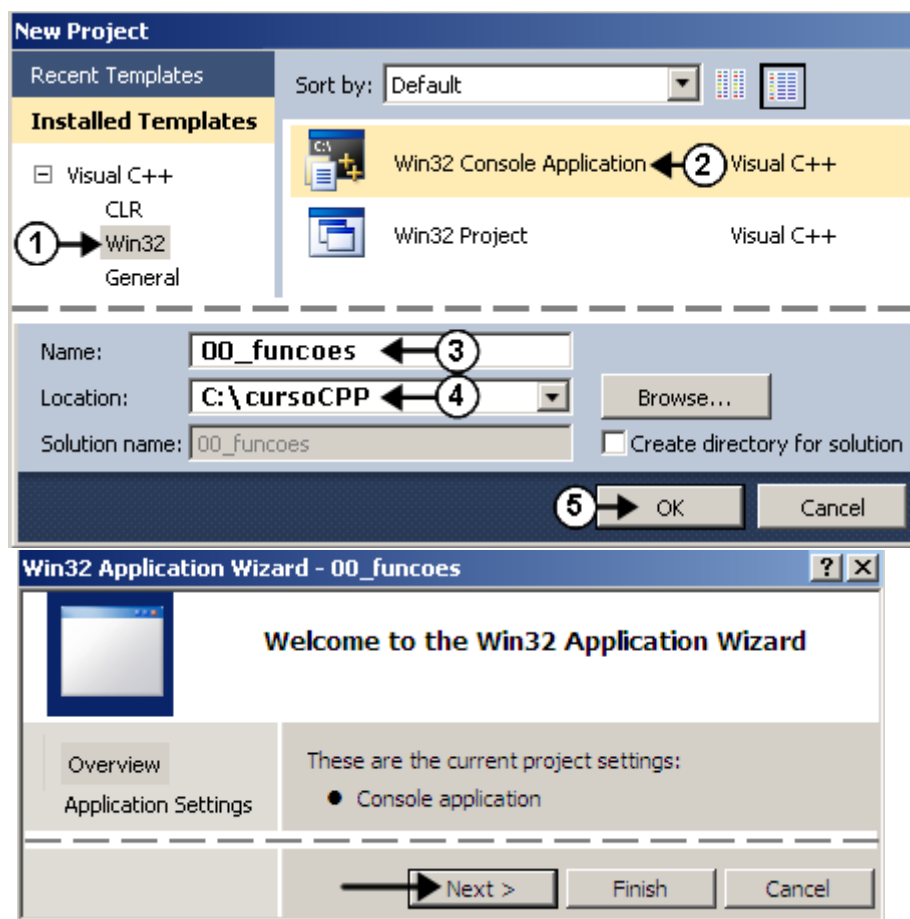
(3) Informe o **nome do projeto**: **00\_funcoes**.

(4) Informe o **diretório** em que será criado: **C:\cursoCPP**.

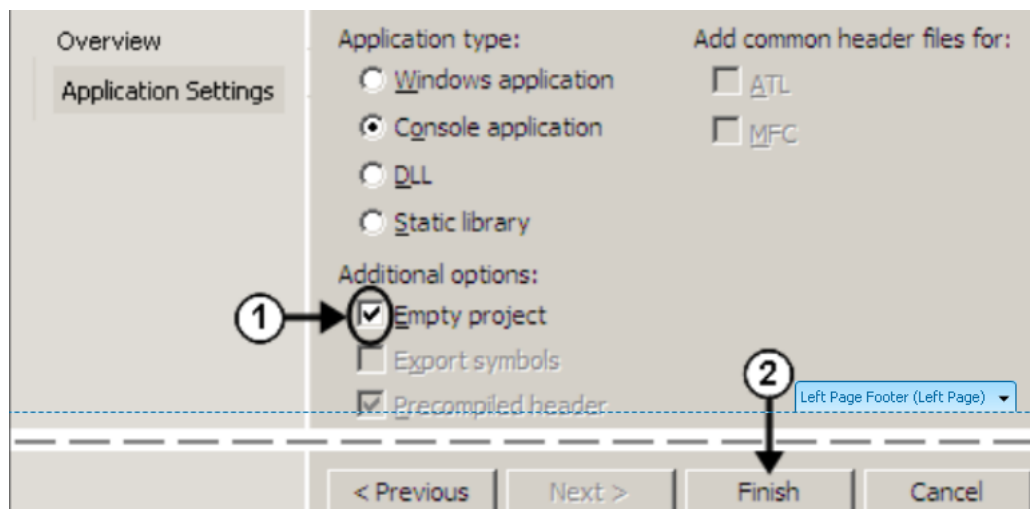
(5) **Prosseguir** para próxima etapa.

Caso ainda não exista, será criado um diretório com o nome do projeto, abaixo do diretório base: **C:\cursoCPP\00\_funcoes**. Caso exista, será usado o diretório existente.

c. Na próxima janela, pressione o botão **Next**, para passar para a próxima etapa:



- d. Finalmente, nas configurações do projeto, faça isto:
- (1) Assinale a opção **Empty project** (projeto vazio).

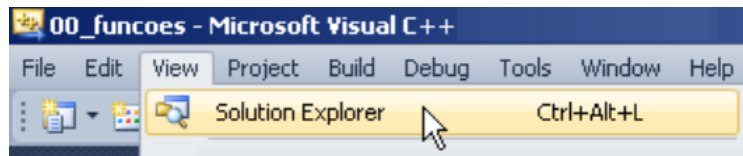




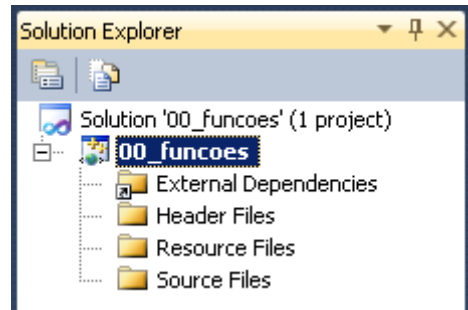
(2) Pressione o botão **Finish** para finalizar a criação do projeto.

e. E estamos de volta à área de trabalho.

Já deverá estar visível o navegador de projetos (aqui chamado *Solution Explorer*). **Caso não esteja**, basta acionar o menu **View**, opção **Solution Explorer**:



f. O **Solution Explorer**, permitirá adicionar arquivos ao projeto e navegar entre eles para edição.



### 2.5.2.2 • Adicionando arquivos ao projeto.

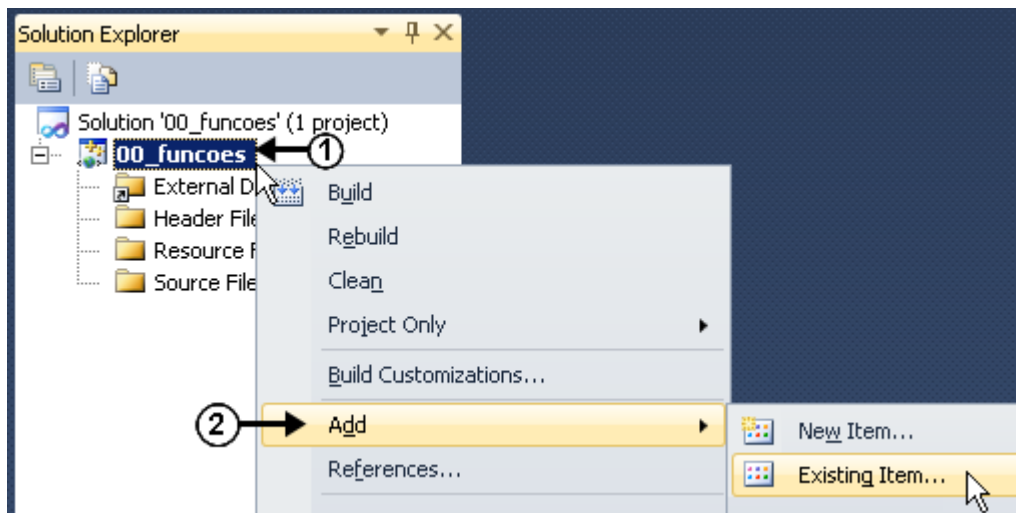
Para acrescentar arquivos temos duas opções:

- **Add / New Item:** acrescenta um novo arquivo.
- **Add / Existing Item:** acrescenta arquivos já existentes.

No nosso caso queremos acrescentar um arquivo já existente: o arquivo **main.cpp** que está no diretório **C:/cursoCPP/00\_funcoes**, conforme é mostrado na segunda figura abaixo. Então, no **Solution Explorer**:

(1) Acione o botão direito do *mouse* sobre o nome do projeto.

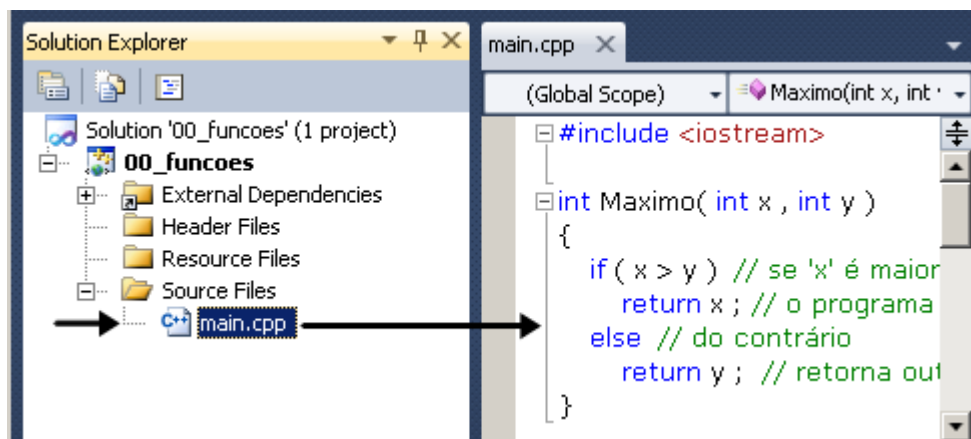
(2) Selecione a opção do *menu* suspenso **Add**, e a sub-opção **Existing Item**.



Adicione o arquivo  
**C:/cursoCPP/  
 00\_funcoes/  
 main.cpp.**



Após adicionar **main.cpp** ao projeto, vemos que o *Solution Explorer* já apresenta um nó para ele. Assim, o arquivo estará sempre diretamente disponível para edição:



### 2.5.2.3 • Compilar e executar.

#### a. Compilar.

Para isso podemos usar a opção de **menu Build**.

Ou então o **botão de atalho Build** que está na barra de ferramentas **Build**:

Caso essa barra **não esteja habilitada**, basta clicar com o botão direito do *mouse* em uma área vazia do *menu*. Aparecerá a lista de barras de ferramentas disponíveis. Então, habilite a barra **Build**:

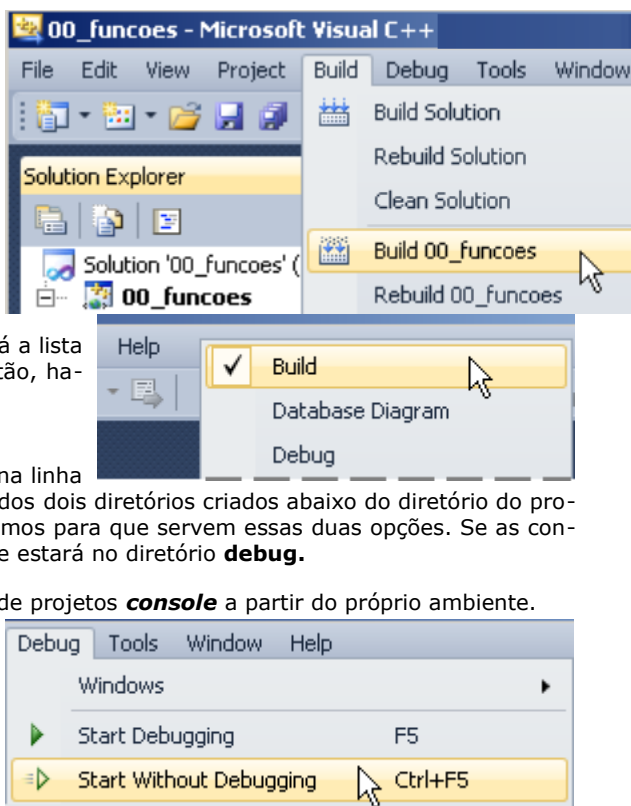
#### b. Finalmente: **executar** o projeto.

Podemos executar o projeto diretamente na linha de comando. O executável estará em um dos dois diretórios criados abaixo do diretório do projeto: **debug** e **release**. Nos anexos, veremos para que servem essas duas opções. Se as configurações *default* não foram alteradas, ele estará no diretório **debug**.

Contudo, o Visual C++ prevê a execução de projetos **console** a partir do próprio ambiente.

Pois, após a execução, irá impor uma pausa à janela *console*, para que seja possível que o seu resultado seja observado.

Basta usar a opção **Start Without Debugging** do *menu Debug*.



## 2.6 • Conclusões sobre o código usado

De tudo o que foi exposto acima, no que diz respeito à **escrita do código**, podemos concluir que qualquer linguagem de programação deve oferecer uma sintaxe e uma gramática para a **escrita de instruções e tomadas de decisão** - (que são os elementos de uma linguagem, designados genericamente como "**statements**").

---

 Em síntese, há **três elementos básicos** indispensáveis para a escrita de programas de computador:

---

**memória;**  
**operações;**  
**fluxo de processamento.**

---

 E, frisando, todos esses elementos devem ser regulados através de uma **sintaxe e uma gramática específicas da linguagem**.

---

Os **elementos básicos** relacionados acima indicam **o que** precisamos escrever para criar programas.

A **sintaxe e a gramática** servem para disciplinar **como** tudo isso deve ser escrito em uma determinada linguagem.

---

Até aqui vimos um breve resumo de alguns desses elementos e de algumas das regras de sintaxe e gramática. Agora, será preciso entender com mais profundidade os conceitos aí envolvidos, bem como os detalhes mais importantes de cada um.

E há também mais outros elementos, que não apareceram nos exemplos mostrados acima.

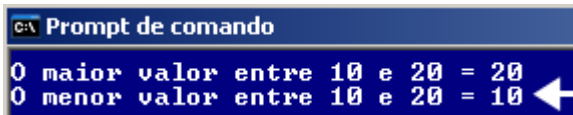
## 2.7 • Revisão do Capítulo 2

### 2.7.1 • Exercício

Tente resolver o exercício abaixo. Em seguida, compare sua solução com a que está no **Anexo B-2-1**, página **427**. Caso não entenda, fale com o instrutor.

**Enunciado:** com base no exemplo da função "**Maximo**", escreva a função "**Minimo**", no arquivo <...>/cursoCPP/00\_funcoes/main.cpp (acima da função "main").

Resultado que deve ser impresso:



```
C:\> Prompt de comando
0 maior valor entre 10 e 20 = 20
0 menor valor entre 10 e 20 = 10
```

**Passos para atingir o objetivo:**

- a. A função "**Minimo**" deve:
  - **Selecionar** o **menor** de dois valores inteiros e **retornar** esse valor.
- b. Em seguida, na função "**main**":
  - **Chame** a função "**Minimo**", passando-lhe os valores apropriados.
  - **Imprima** o resultado (retorno) devolvido pela função "**Minimo**".
- c. **Compilar**, usando **ambientes de desenvolvimento**, ou chamando, **na linha de comando, uma** das duas linhas abaixo:
 

```
cl /EHsc /Fe"00_funcoes" main.cpp // compilador microsoft - Windows
g++ main.cpp -o 00_funcoes // compilador gcc - Windows ou Unix/Linux
```

  - Se ocorreram **erros de compilação**, verifique e corrija as linhas de erro.
    - Dúvidas? Reveja as dicas sobre identificação de erros (seção **2.4**, página **54**, acima). E as regras **básicas de gramática** (seção **2.2.4**, página **46**, acima).
- d. **Execute** o programa, chamando **00\_funcoes** (*Windows*) ou **./00\_funcoes** (*Unix/Linux*) na linha de comando. Observe se funciona corretamente.



Compare o que você fez com a solução da apostila: página **427**.

## 2.7.2 • Questões para revisão

Responda às questões abaixo, assinalando **todas** as respostas corretas (**uma ou mais**). **Em seguida, compare suas respostas** com as respostas localizadas no **Anexo B-2-2, página 428**. **Caso não entenda**, encaminhe as dúvidas ao instrutor.

**Cap.2 - 1.** A respeito de **funções**, assinale as afirmações corretas:

- a. ☐ Uma **função** é o mesmo que uma linha de instrução.
- b. ☐ Uma função é um bloco contendo um conjunto de linhas de instrução **podendo ou não** ter um nome.
- c. ☐ Uma função é um bloco contendo um conjunto de linhas de instrução, e obrigatoriamente deve ter um **nome** que a identifique.
- d. ☐ Nomes de função são seguidos **obrigatoriamente** por parênteses .
- e. ☐ O bloco de instruções de uma função é **iniciado** pelo próprio nome da função e **encerrado** com a instrução **return**.
- f. ☐ O bloco de instruções de uma função é **iniciado** e **encerrado** com chaves: { ... }
- g. ☐ Se uma **função** tiver apenas uma **única linha de instrução**, as chaves podem ser **omitidas**.
- h. ☐ A expressão [ **c = Maximo ( a , b ) ;** ] contém uma **chamada** de função.

**Cap.2 - 2.** Sobre **linhas de instrução**, assinale as afirmações corretas:

- a. ☐ Uma linha de instrução é encerrada pela quebra de linha do editor de textos.
- b. ☐ Uma linha de instrução é encerrada com um ponto e vírgula.

**Cap.2 - 3.** Considerando o código abaixo, assinale as afirmações corretas:

```
int x , y ;  
// ...  
if ( x > y )  
std::cout << "x é maior que y" << "\n";  
std::cout << "agora vou encerrar\n" ;
```

- a. ☐ **Apenas** se "x" for maior que "y", será impresso **x é maior que y** e **agora vou encerrar**.
- b. ☐ **Nada** será impresso.
- c. ☐ **Apenas** se "x" for maior que "y", será impresso **x é maior que y**. Em qualquer caso, **sempre** será impresso **agora vou encerrar**.
- d. ☐ **Sempre** será impresso **x é maior que y** e **agora vou encerrar**.

**Cap.2 - 4.** Uma aplicação tem seu **início** (ou ponto de entrada) em:

- a. ☐ Nesta linha: [ **#include <iostream>** ]
- b. ☐ Na primeira função que esteja escrita em um **arquivo** que deve ter o nome de **main.cpp**.

- 
- c. ☐ Em uma **função** que deve ter o nome especial de **main**, não importando o nome do arquivo em que esteja escrita. Essa função é obrigatória e única em qualquer aplicação diretamente executável.
- 

**Cap.2 - 5.** A respeito do **compilador** podemos afirmar que:

- 
- a. ☐ O compilador serve **apenas** para analisar se o código foi escrito corretamente segundo as regras da linguagem.
- 
- b. ☐ Ele analisa se o código foi escrito corretamente, de acordo com a linguagem, e, em caso positivo, irá traduzi-lo para linguagem de máquina.
- 
- c. ☐ O compilador indica erros de escrita, mas **não fornece informações** sobre esse erro.
- 
- d. ☐ O compilador indica erros de escrita, emitindo uma mensagem sobre o tipo do erro e a linha em que ele ocorreu. **Devemos usar essas duas informações** para corrigir os erros mais rapidamente.
- 



**Compare o que você fez com a solução da apostila: página 428.**

## • Capítulo 3

### ▪ Programação: princípios básicos

Neste capítulo, teremos uma visão geral sobre **operações, memória e fluxo de processamento, conceituando** o que vimos no capítulo anterior.

Além disso, veremos um novo recurso de fluxo de processamento: os **laços**, para a repetição da execução de determinadas instruções.

Quanto à implementação em **C++**, além de uma associação mais direta dos recursos da linguagem aos princípios e conceitos, teremos também novos exemplos.

3.1 •	Tópicos de destaque neste capítulo.....	72
3.2 •	Operações, memória e fluxo de processamento.....	73
3.2.1 ▪	Operações.....	73
3.2.2 ▪	Memória.....	73
3.2.3 ▪	Fluxo de processamento.....	74
3.2.4 ▪	Conceitos de Verdadeiro e Falso.....	75
3.2.5 ▪	Implementação em C++.....	77
3.2.5.1 ▪	Operações e operadores.....	77
3.2.5.2 ▪	Memória.....	77
3.2.5.3 ▪	Escrevendo código.....	78
3.2.5.4 ▪	Verdadeiro e Falso.....	79
3.3 •	Fluxo de processamento: laços.....	80
3.3.1 ▪	Princípios.....	80
3.3.2 ▪	Implementação em C++.....	82
3.3.2.1 ▪	O controle de laço "while".....	82
3.3.2.2 ▪	Compilando e executando.....	86
3.3.2.3 ▪	Olaço for.....	86
3.3.3 ▪	O tipo bool e o resultado de avaliações.....	89
3.4 •	Comentários.....	89
3.5 •	Conclusão.....	90
3.6 •	Revisão do Capítulo 3.....	91
3.6.1 ▪	Exercício.....	91
3.6.2 ▪	Questões para revisão.....	94

## 3.1 • Tópicos de destaque neste capítulo.

### ➤ Entendendo os princípios básicos de programação

- ◆ Linhas de instrução e operações.
- ◆ Memória: reservar áreas da memória para que possam ser acessadas nas operações a executar.
- ◆ Fluxo de processamento: sequência de instruções, tomadas de decisão e desvios.
- ◆ Tomadas de decisão: conceitos de verdadeiro e falso.
- ◆ Fluxo de processamento: laços para a repetição da execução de um determinado bloco de instruções.

### ➤ Implementação em C e C++: associando os exemplos anteriores aos conceitos expostos acima.

- ◆ Os recursos das linguagens C e C++ já vistos nos exemplos do capítulo anterior (funções, instruções, operações, reserva de memória) serão aqui relacionados aos princípios e conceitos básicos de programação.

### ➤ Implementação em C e C++: novos elementos.

- ◆ conceito de verdadeiro e falso em C ou C++;
- ◆ implementação de laços em C ou C++;
- ◆ entrada de dados fornecidos pelo usuário.



## 3.2 • Operações, memória e fluxo de processamento

### 3.2.1 • Operações

A maior parte das instruções de um programa de computador é constituída de **operações**, identificadas por uma palavra ou um símbolo de **operador**. *Alguns exemplos:*

- **cópia de valor** (armazena um valor): atribuição(=)
- operações **aritméticas**: soma(+), subtração(-), multiplicação(\*), divisão(/), etc.
- operações **relacionais**: maior-que(>), menor-que(<), igualdade(==), etc.


**Exemplo (uma operação de venda):**

- a) **armazene** o valor 100 como "**valor-do-produto**";
- b) **armazene** o valor 10 como "**valor-do-imposto**";
- c) **armazene** o valor 110 como o "**valor-pago**";
- d) **some** o "**valor-do-imposto**" ao "**valor-do-produto**";
- e) **armazene** o resultado da soma em "**valor-da-venda**";
- f) **compare**: "**valor-pago**" é menor-que (<) "**valor-da-venda**" ?

Para atingir a operação completa (uma venda), temos aí **três diferentes operações**:

- **armazenamento** (operação de atribuição);
- **soma** (operação aritmética);
- **comparação** (operação relacional).

A **primeira questão** a considerar aí diz respeito à operação de **armazenamento**: armazenar algo pressupõe que guardamos alguma coisa em algum lugar.


 E, em um programa de computador, o local onde armazenamos valores é a **memória**.

A **segunda questão** a considerar no exemplo acima é que as instruções seguem uma **sequência lógica** (de "a" até "f").

 Logo, temos um **fluxo de processamento**.

### 3.2.2 • Memória

Qualquer linguagem de computador oferece algum meio para **acessar a memória**, visando a **armazenar** e a **recuperar** valores. As **operações** trabalham com valores e estes devem existir em algum lugar da memória do computador.

 Para isso, uma linguagem precisa definir meios para **reservar memória**, de modo que depois ela possa ser acessada.

E isso pode ser feito de muitas maneiras, algumas melhores do que outras.

Por exemplo, há linguagens que providenciam a **reserva** de memória sempre que é executada uma nova operação de armazenamento.

Ou seja, o programador não precisa se preocupar com a reserva em si. Isso parece bom, mas tem um preço em termos de segurança e eficiência (falaremos disso mais a frente).

E, como logo veremos, em C e C++ teremos algumas maneiras de reservar memória. O programador deverá escolher a maneira mais adequada para cada caso.

### 3.2.3 • Fluxo de processamento

No exemplo acima, vimos que há uma **sequência de instruções**, ou seja, um **fluxo de processamento**. E, nesse exemplo, o **fluxo é puramente sequencial** ou linear, pois todas as instruções são executadas ordenadamente de “a” até “f”.

Em um programa real temos às vezes longos trechos com instruções que devem ser executadas desse modo puramente sequencial. Mas temos também, em diversos momentos, a necessidade de interromper essa sequência linear, estabelecendo **desvios de fluxo**.

Retomando o próprio exemplo acima (onde paramos na operação “f”), podemos perceber que **falta** alguma coisa:

**f) compare: “valor-pago” é menor-que (<) “valor-da-venda” ?**

Bem, e daí? **O que fazer após essa comparação?** Após a comparação, será preciso **tomar decisões** em função do resultado dessa operação.

Então, deveríamos acrescentar o seguinte:

**f.1) caso o “valor-pago” seja menor que (<) o “valor-da-venda”**

**→ emitir um alarme ;**

**f.2) do contrário → completar a venda ;**

**f.2.1) se o caminho seguido foi este último, então compare:**  
**“valor-pago” é maior-que (>) “valor-da-venda” ?**

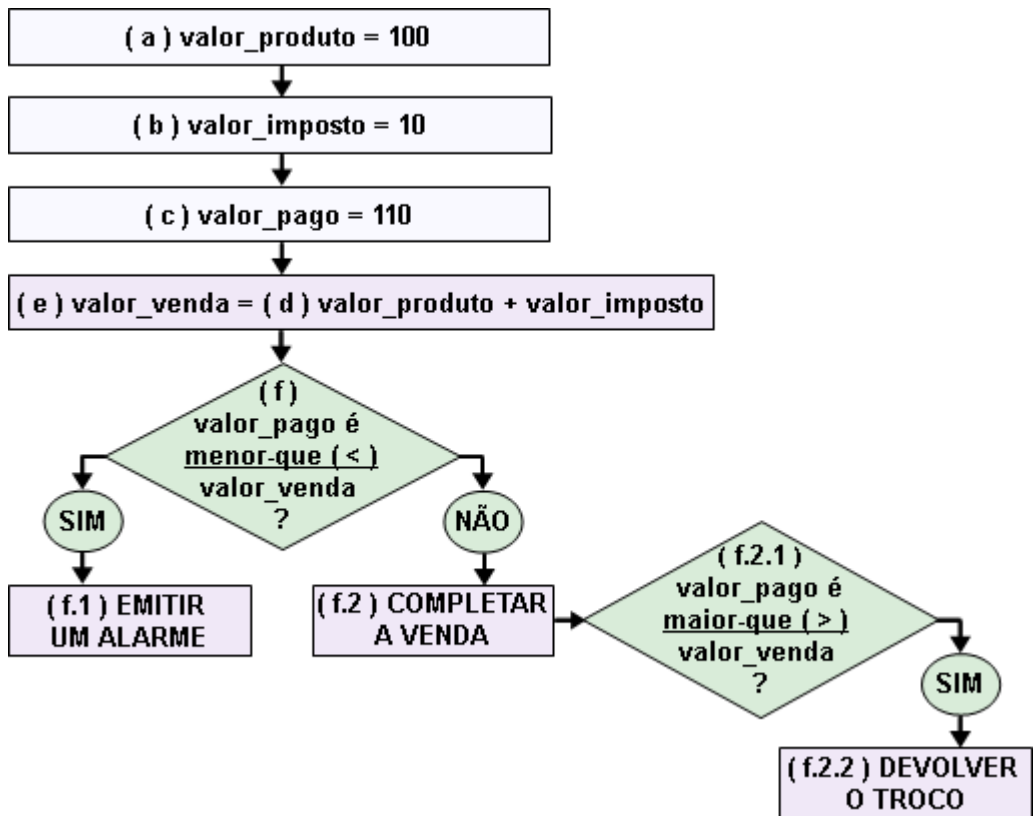
**f.2.2) caso seja, → devolver o troco .**

☞ Isso significa que, em consequência das comparações, as instruções seguintes serão executadas mediante **desvios no fluxo**.

Após a primeira comparação (“f”), temos **dois caminhos**: “*emitir um alarme*” ou “*completar a venda*”. Ou uma coisa ou a outra.

- Assim, como **apenas uma** dessas duas operações será executada, a **sequência** linear de instruções (sempre executadas uma após a outra) **será quebrada** nesse ponto: ou “*emitir um alarme*”, ou “*completar a venda*”.
- Além disso, se o caminho a seguir for “*completar a venda*”, será preciso avaliar ainda se o “valor-pago” está **maior que** o “valor-da-venda”.
  - Se estiver, será preciso “*devolver o troco*”, o que implica em um **novo desvio** para que seja seguido **este caminho** (pois se os dois valores estiverem iguais, não haverá nada a devolver e o caminho será outro).

**Essa situação pode ser melhor visualizada no seguinte fluxograma:**



🔗 Temos, aí, em síntese, tanto instruções que são executadas sequencialmente, como **tomadas de decisão** seguidas por **desvios**.

Portanto, um **fluxo de processamento** é constituído por **sequências de instruções, tomadas de decisão e desvios**.

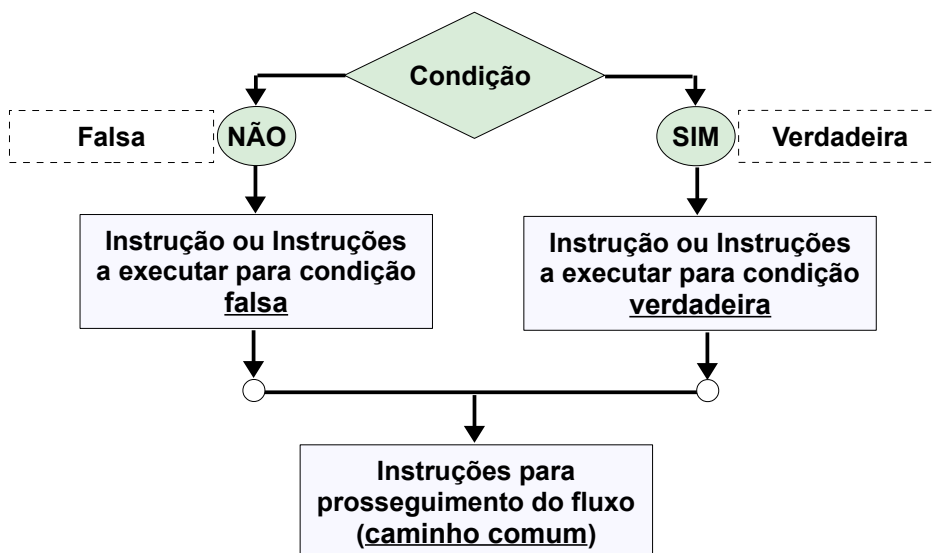
### 3.2.4 • Conceitos de Verdadeiro e Falso

O princípio a extrair do fluxo acima e de suas conclusões é que:

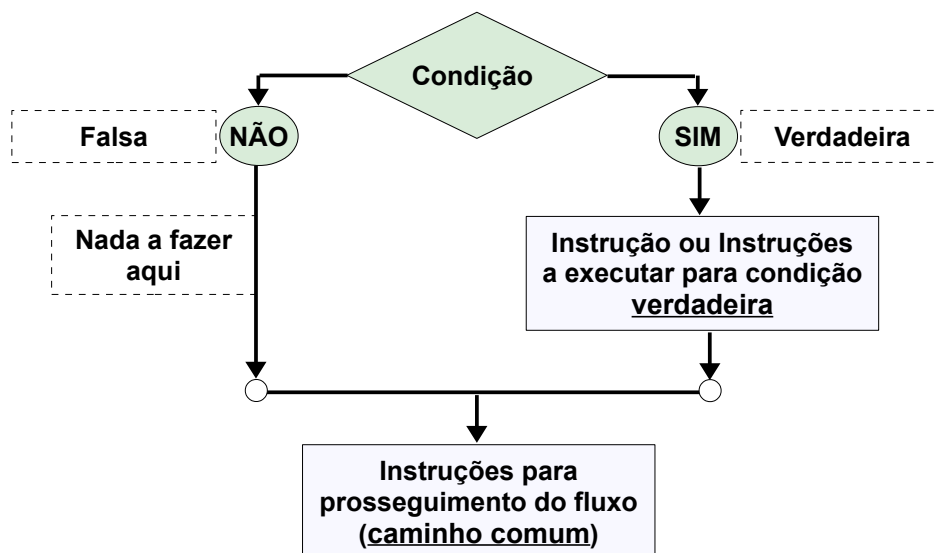
- Uma tomada de decisão é baseada no conceito "**verdadeiro X falso**".
- Assim, tomar uma decisão significa avaliar se uma determinada **condição é verdadeira**.
  - Se for **verdadeira**, um determinado caminho (instrução ou grupo de instruções) será executado.
  - E se **não** for **verdadeira** (condição avaliada como **falsa**), esse caminho não será executado.
    - No lugar dele, será executado um outro caminho (outra instrução ou grupo de instruções).
    - Ou, simplesmente, a instrução ou grupo de instruções associados à avaliação da condição como verdadeira serão pulados, e o processamento seguirá diretamente pelo caminho posterior, comum às duas alternativas.

**Exemplos:**

- a. Caminhos diferentes para as avaliações "verdadeira" e "falsa".  
Processamento comum em seguida.



- b. Caminho diferente apenas para a avaliação "verdadeira".  
Processamento comum em seguida.



### 3.2.5 • Implementação em C++

#### 3.2.5.1 • Operações e operadores

Precisaremos de áreas de memória para executar certas **operações**. Portanto são estas que realizam a computação propriamente dita, fazendo as coisas funcionarem.

Em C ou C++ operações são identificadas por seus **operadores**, que são representados através de símbolos ou de palavras reservadas.

Inicialmente, precisamos apenas de um pequeno conjunto dos operadores existentes nessas linguagens:

Alguns operadores:		
<b>Operadores aritméticos:</b>		
-	<b>Subtração</b>	[ x - y ]; subtrai 'y' de 'x'
+	<b>Soma</b>	[ x + y ]; soma 'x' a 'y'
*	<b>Multiplicação</b>	[ x * y ]; multiplica 'x' por 'y'
/	<b>Divisão</b>	[ x / y ]; divide 'x' por 'y'
<b>Operadores relacionais (comparações):</b>		
>	<b>Maior que</b>	[ x > y ]; 'x' é maior-que 'y' ?
>=	<b>Maior que ou igual</b>	[ x >= y ]; 'x' é maior-ou-igual-a 'y' ?
<	<b>Menor que</b>	[ x < y ]; 'x' é menor-que 'y' ?
<=	<b>Menor que ou igual</b>	[ x <= y ]; 'x' é menor-ou-igual-a 'y' ?
==	<b>Igualdade (<i>equal</i>)</b>	<u>Obs.:</u> um duplo "=" [ x == y ]; 'x' é igual-a 'y' ?
!=	<b>Desigual (<i>not equal</i>)</b>	[ x != y ]; 'x' é diferente-de 'y' ?
<b>Outros operadores:</b>		
=	<b>Atribuição</b>	<u>Obs.:</u> um único "=" [ x = y ]; copia 'y' para 'x'
<f> ()	<b>Os parênteses representam aí uma <i>chamada à função</i> "f"</b>	

Para a implementação dos fluxos exibidos acima, iremos usar alguns desses operadores. Mais a frente essa tabela irá crescer, conforme as requisições das novas situações e exemplos.

#### 3.2.5.2 • Memória

Em exemplo do capítulo anterior (seção 2.2.2, página 43) tínhamos:


```
int a , b , c ;
```

o que significa que solicitamos a reserva de memória para três valores inteiros.

Se fossemos fazer o mesmo para a "operação de venda" exposta acima, teríamos algo como:

<code>int valor_produto ;</code>	"int": reserva memória para um número inteiro...
<code>int valor_imposto ;</code>	idem
<code>int valor_pago ;</code>	idem
<code>int valor_venda ;</code>	idem

Isso significa que, em C ou C++, um pedido de reserva de memória deve indicar para qual finalidade essa memória será usada.

 Veremos adiante (capítulo 4, página 97) qual a **razão de ser** desse requisito.

Por enquanto, basta frisar que a finalidade prevista para o uso das quatro memórias criadas na tabela acima é **armazenar números inteiros**. E isso significa que uma reserva de memória deve indicar um **tipo** para os dados a armazenar. Pode-se objetar, com razão, que números inteiros não parecem muito apropriados para representar valores que, provavelmente, precisarão de uma **parte fracionária** (centavos). Mas, como veremos no capítulo já referido, o "**int**" não é o único **tipo** de dados suportado pela linguagem; poderemos, portanto, escolher algum outro tipo para valores dessa natureza.

Neste momento, o **foco é: a reserva de áreas de memória** em C ou C++, deve ser feita com a indicação de um determinado **tipo** que as **qualifique**.

Com base nisso, poderemos entender o que será feito no exemplo de implementação da "operação de venda" (abaixo). Nesse exemplo, teremos a reserva de memórias tal como foi descrito aqui, bem como as operações a realizar com os dados nelas armazenados e, finalmente, as decisões a tomar em função da lógica do problema - usando-se uma implementação típica em C ou C++.

3.2.5.3 • Escrevendo código

Vamos ver agora como escrever código em C ou C++, para implementar o primeiro exemplo de fluxo exposto acima (a "operação de venda").

```
// (a) Reserve um novo local de memória, que acessarei usando o apelido
// "valor_produto". Em seguida, armazene 100 nesse local de memória:
int valor_produto = 100 ; // "int": reserva para um número inteiro...

// (b) Reserve um novo local de memória, que acessarei usando o apelido
// "valor_imposto". Em seguida, armazene 10 nesse local de memória:
int valor_imposto = 10 ; // "int": idem...
```

// (c) **Reserve** um novo local de **memória**, que acessarei usando o apelido "valor\_pago". Em seguida, **armazene** 110 nesse local de memória:

```
int valor_pago = 110 ; // "int": idem...
```

// (d) **Some** os valores armazenados nas memórias "valor\_produto" e "valor\_imposto". **E, em seguida:**

// (e) **Reserve** um novo local de **memória**, que acessarei usando o apelido "valor\_venda". Em seguida, **armazene aí** o resultado da soma:

```
int valor_venda = valor_produto + valor_imposto ; // "int": idem.
```

// (f) **Compare:** o "valor\_pago" é **menor que (<)** o "valor\_venda" ?

```
if ( valor_pago < valor_venda )
{
    // (f.1) ... escrever aqui o código para "emitir um alarme" ...
}
else
{
    // (f.2) ... escrever aqui o código para "completar a venda" ...

    // (f.2.1) compare: "valor_pago" é maior que (>) "valor_venda" ?
    if ( valor_pago > valor_venda )
    {
        // (f.2.2) ... escrever aqui o código para "devolver o troco" ...
    }
}
```

### 3.2.5.4 • Verdadeiro e Falso

No exemplo de código acima temos duas **avaliações de condição**:

[ **if (valor\_pago < valor\_venda)** ] e [ **if (valor\_pago > valor\_venda)** ]. A avaliação de uma condição tem um resultado que pode ser verdadeiro ou falso. Em função desse resultado, um caminho de processamento é seguido.

Generalizando, em C e C++, os conceitos de verdadeiro e falso seguem um princípio muito simples:



**Zero é falso.**

Tudo o que é diferente de falso é verdadeiro.

Logo, **tudo o que é diferente de zero é verdadeiro.**

Desse modo, uma **expressão**:

- É **falsa** se a sua **avaliação retorna zero**.
- É **verdadeira** se **retorna qualquer valor diferente de zero**.



Em **C++**, como veremos mais abaixo, diferentemente de **C**, temos o tipo **bool**.

E o resultado de uma avaliação é sempre do tipo **bool**, assumindo os valores **true** (se verdadeira) e **false** (se falsa).

**Exemplos:****a. Caminhos diferentes para as avaliações "verdadeira" e "falsa":**

```
int x , y ;
// ...
if ( x==y ) // [se x for igual a y] retorna 1 (true em C++) se verdadeiro
    std::cout << "condição verdadeira: valor de 'x' é igual a 'y' \n" ;
else
    std::cout << "condição falsa: valor de 'x' é diferente de 'y' \n" ;
std::cout << "o processamento comum prossegue aqui \n" ;
```

**b. Caminho diferente apenas para a avaliação "verdadeira":**

```
int x , y ;
// ...
if ( x == y ) // [se x for igual a y] retorna 1 (true em C++) se verdadeiro
    std::cout << "condição verdadeira: valor de 'x' é igual a 'y' \n" ;
// neste caso, não há nada de especial a fazer se a condição for falsa.
std::cout << "o processamento comum prossegue aqui \n" ;
```

Seguindo o princípio "**falso se zero, verdadeiro se diferente de zero**", podemos simplesmente **testar o valor** de uma determinada memória (variável):

**a. Caminhos diferentes para as avaliações "verdadeira" e "falsa":**

```
int x ;
// ...
if ( x ) // retorna 1 (true em C++) – verdadeiro, se "x" estiver diferente de zero
    std::cout << "verdadeiro: valor de 'x' é diferente de zero \n" ;
else
    std::cout << "falso: valor de 'x' é zero \n" ;
std::cout << "o processamento comum prossegue aqui \n" ;
```

**b. Caminho diferente apenas para a avaliação "verdadeira".**

```
if ( x ) // retorna 1 (true em C++) – verdadeiro, se "x" estiver diferente de zero
    std::cout << "verdadeiro: valor de 'x' é diferente de zero \n" ;
// neste caso, não há nada de especial a fazer se a condição for falsa.
std::cout << "o processamento comum prossegue aqui \n" ;
```

## 3.3 • Fluxo de processamento: laços

### 3.3.1 • Princípios

O exemplo de fluxo mostrado acima não é muito útil para um programa do "mundo real", pois ele só permite a execução de **uma única operação de venda**.



Em um programa real, provavelmente essas instruções deveriam ser **repetidas** para que **diversas vendas** fossem processadas.

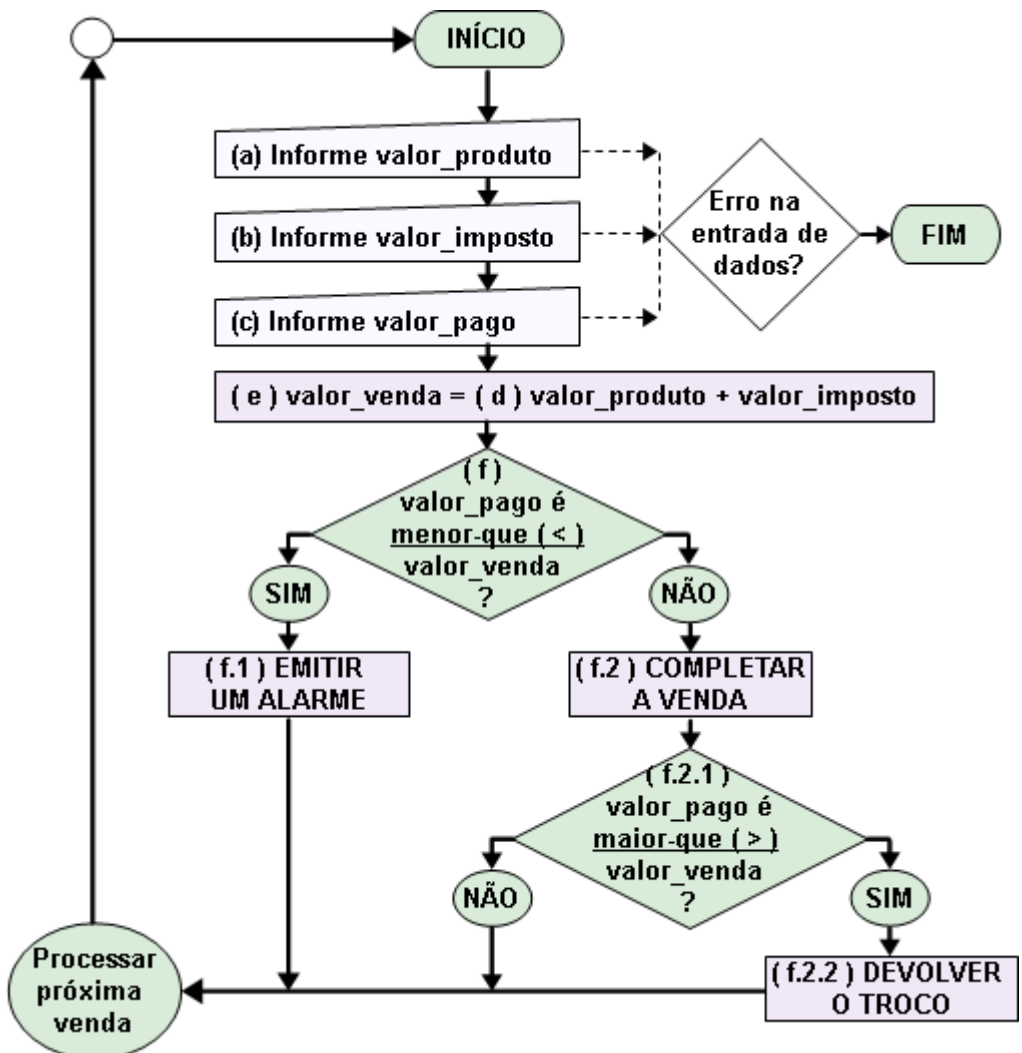
Para isso temos um outro elemento básico de programação: os **laços**, que são estruturas de controle que permitem a **repetição de um grupo de instruções**.

Além disso, não faria sentido prático **repetir** a operação sempre com os **mesmos valores**. Então, devemos substituir a forma como eles são estabelecidos. Ao invés de valores constantes para "**valor\_produto**", "**valor\_imposto**" e "**valor\_pago**", devemos ter uma entrada de dados (leitura de um arquivo, ou informações prestadas pelo usuário).

É óbvio que ainda não é o momento de escrever programas para o "mundo real". Mas vamos nos aproximar um pouquinho de uma situação mais realista.

**Desse modo, o fluxo exposto acima deveria ser alterado da forma exibida abaixo.**

No novo fluxo, após uma operação completa de venda, essa operação será **repetida**, para que seja processada uma **próxima venda**.



### 3.3.2 • Implementação em C++

No fluxo que vemos acima, ao final de uma operação completa de venda, o processamento **volta ao início**.

Além disso, nos três primeiros passos (de "a" até "c"), ao invés de definirmos os valores de "**valor\_produto**", "**valor\_imposto**" e "**valor\_pago**" com as constantes **100**, **10** e **110**, pedimos ao usuário que os informe. Assim, cada venda terá valores diferentes.

Para a **implementação** em C ou C++ , já sabemos que é necessário que:

- O código esteja dentro de uma **função** - e deve existir uma função **main**, para que o programa tenha um ponto de início.

Além disso, neste exemplo:

- Precisaremos usar um **controlador de laço**.
- E, também, instruções de **entrada de dados**.

Há três controladores de laço que podemos usar. Em princípio, **qualquer um** poderia ser usado aqui. Mas se existem três é por uma razão: cada um deles se ajusta melhor a uma determinada situação.

- Existem situações em que há uma **progressão** até que um determinado **limite** seja atingido, e para isso teremos o laço **for** (veremos exemplos mais abaixo).
- Já para a situação que precisamos implementar aqui (o fluxo exposto acima), não temos propriamente uma progressão, mas uma **solicitação contínua** de dados ao usuário, os quais são processados e avaliados. Ou seja, o programa continuará pedindo dados ao usuário, para que uma nova venda seja processada. Caso ocorra um erro nessa entrada de dados, então o programa encerra.  
Para esta situação temos outros dois laços. Um deles é o **while**.

#### 3.3.2.1 • O controle de laço "while"

O laço **while** analisa uma **condição** inicialmente e permite executar instruções repetidamente, voltando sempre à análise da condição, **até que a condição torne-se falsa** - e então o laço é **interrompido**.

Em certos casos, em que a interrupção do laço é determinada por uma condição especial que só irá ocorrer durante o próprio processamento das instruções a repetir, podemos inclusive estabelecer um **laço infinito** - e sua **interrupção** será feita por um **desvio**.

**Exemplos:**

A condição pode mudar a qualquer momento	Condição imutável (laço infinito)
<pre>while (x &gt; 1) // a <b>condição</b> pode ser {           // <b>verdadeira ou falsa</b>     // Só executa se     // a condição é <b>verdadeira</b>.     // O valor de '<b>x</b>' pode mudar     // <b>influindo na próxima avaliação</b>     // da <b>condição</b>. }</pre>	<pre>while ( 1 ) // "<b>1</b>" é <b>sempre verdadeiro</b>... {     // Executa <b>sempre</b>.     // Ocorrerá, em algum momento, uma     // situação especial (por exemplo, erro     // em uma entrada de dados), que     // exigirá a <b>interrupção</b> do laço <b>aqui</b>. }</pre>

No exemplo à direita, a avaliação é sempre verdadeira, pois o que é avaliado é a **constante 1** que é imutável (e **1** é verdadeiro, já que é diferente de zero).

Então, a **implementação do fluxo exibido acima (um laço com múltiplas operações de venda)** poderia ser feita do seguinte modo:

## 1) Resultados que devem ser impressos:

<pre> C:\ Prompt de comando -Digite numeros. Para interromper, digite algo invalido &lt;uma letra&gt;: -(a) Informe o valor do produto: 100 -(b) Informe o valor do imposto: 10 -(c) Informe o valor pago: 110  *** Completar venda  -Digite numeros. Para interromper, digite algo invalido &lt;uma letra&gt;: -(a) Informe o valor do produto: A fim de programa         </pre>	<div style="border: 1px solid black; padding: 5px; width: fit-content;">a) Valor pago igual-a valor do produto + valor do imposto</div> <div style="margin-top: 20px;"> <div style="border: 1px dashed black; padding: 5px; width: fit-content;">Valores informados pelo usuário</div> <div style="margin-top: 20px;"> <div style="border: 1px dashed black; padding: 5px; width: fit-content;">Usuário informou entrada incorreta</div> </div> </div>
<pre> C:\ Prompt de comando -Digite numeros. Para interromper, digite algo invalido &lt;uma letra&gt;: -(a) Informe o valor do produto: 100 -(b) Informe o valor do imposto: 10 -(c) Informe o valor pago: 120  *** Completar venda *** Devolver o troco = 10  -Digite numeros. Para interromper, digite algo invalido &lt;uma letra&gt;: -(a) Informe o valor do produto: A fim de programa         </pre>	<div style="border: 1px solid black; padding: 5px; width: fit-content;">b) Valor pago maior-que valor do produto + valor do imposto</div>
<pre> C:\ Prompt de comando -Digite numeros. Para interromper, digite algo invalido &lt;uma letra&gt;: -(a) Informe o valor do produto: 100 -(b) Informe o valor do imposto: 10 -(c) Informe o valor pago: 100  *** Valor pago: menor. Faltou = 10  -Digite numeros. Para interromper, digite algo invalido &lt;uma letra&gt;: -(a) Informe o valor do produto: A fim de programa         </pre>	<div style="border: 1px solid black; padding: 5px; width: fit-content;">c) Valor pago menor-que valor do produto + valor do imposto</div>

## 2) Código fonte:

```

#include <iostream>
int main()

```

```

{
    int valor_produto , valor_imposto, valor_venda, valor_pago ;

    while ( 1 )
    {
        // (a), (b), (c): solicita ao usuário que informe os três valores:
        std::cout << "\n-Digite numeros. Para interromper, "
                    << "digite algo invalido (uma letra):\n\n" ;

        std::cout << "-(a) Informe o valor do produto: ";
        std::cin >> valor_produto ; // "cin" pegará uma entrada no teclado
                                    // e a copiará para "valor_produto".

        if ( std::cin.fail ( ) ) // se houve uma entrada incorreta...
            break ; // interrompe o laço.

        std::cout << "-(b) Informe o valor do imposto: ";
        std::cin >> valor_imposto; // idem

        if ( std::cin.fail ( ) ) // se houve uma entrada incorreta...
            break ; // interrompe o laço.

        std::cout << "-(c) Informe o valor pago: ";
        std::cin >> valor_pago; // idem

        if ( std::cin.fail ( ) ) // se houve uma entrada incorreta...
            break ; // interrompe o laço.

        std::cout << "\n"; // quebra linhas...

        // (d) soma "valor_produto" e "valor_imposto";
        // (e) armazena o resultado em "valor_venda"
        valor_venda = valor_produto + valor_imposto ;

        // (f) compara: "valor_pago" é menor-que "valor_venda"?
        if ( valor_pago < valor_venda )
            std::cout << "*** Valor pago: menor. Faltou = "
                        << valor_venda - valor_pago << "\n";

        else
        {
            std::cout << "*** Completar venda\n";
            // Falta fazer: chamar aqui a função "CompletarVenda()" ...
            if ( valor_pago > valor_venda )
            {
                std::cout << "*** Devolver o troco = "
                            << valor_pago - valor_venda << "\n";
                // Falta fazer: chamar aqui a função "DevolverTroco()" ...
            }
        }
    }
} // fim do laço while.

std::cout << "\nfim de programa\n" ;
return 0;
} // fim da função main.

```

Executa o bloco abaixo **enquanto** a condição entre **parênteses** for verdadeira. Neste caso, executará **para sempre**, pois "1" é verdadeiro. E é uma constante - que não será alterada.

Acima introduzimos novos elementos. um laço **while** com um desvio **break** e **std::cin**.

- **while:** é um controle de laços que é seguido por uma **condição entre parênteses**.
  - Se e enquanto a condição for **verdadeira**, as instruções associadas ao controle de laço serão executadas **repetidamente**.

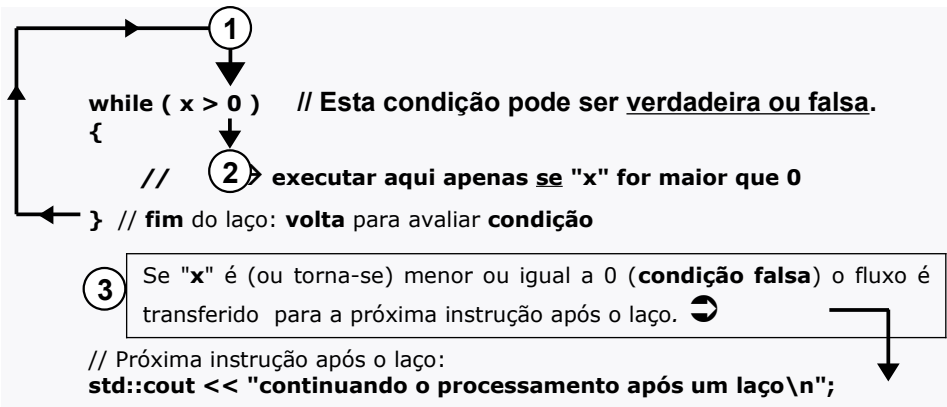
- Para associar instruções precisamos criar um **bloco de código** (iniciado e encerrado com **chaves**), que indica quais são as instruções que devem ser repetidas enquanto a condição for verdadeira. **Exemplo:**

```
int x;
// ....
while ( x > 0 )
{
    // uma ou mais instruções a executar enquanto "x" for maior que 0
}
```

- Caso exista apenas uma instrução as chaves podem ser **omitidas**:

```
while ( x > 0 )
    // uma única instrução a executar enquanto "x" for maior que 0
```

- Um laço **while** é **interrompido** quando a **condição é falsa**.



- Um laço pode ser **interrompido** com o *statement* de **desvio break**. Um **break** provoca um salto para a próxima instrução após o final do laço:

```
while ( x > 0 )
{
    int condicao_especial ;
    // .....
    if ( condicao_especial )
        break ; // Interrompe... ➡
    // ...
}
// Próxima instrução após o laço:
std::cout << "continuando o processamento após um laço\n";
```

- **cin**: é um recurso da biblioteca padrão de **C++** que permite aguardar por uma entrada de dados na entrada padrão do computador (normalmente o teclado).
  - a entrada será armazenada nas memórias (variáveis) que seguem o operador ">>".

Exemplos:

```
int x, y, z;
std::cin >> x ;    // aguarda por uma entrada de dados
                    // que será armazenada em "x"

std::cin >> x >> y >> z ; // aguarda por três entradas de dados
                    // que serão armazenadas, respectivamente, em "x", "y" e "z".
```

O que pode ser sintetizado assim:

entrada	vai para →	memória		
std::cin	>>	x	y	z

3.3.2.2 • Compilando e executando

Para fixar o que vimos acima, vamos criar um projeto e testar o código da "operação de venda com laço" que acabamos de descrever. Para isso, siga os passos abaixo:

- a. **Criar um arquivo** (usando **ambientes de desenvolvimento ou um editor de textos**).
  - Dentro do diretório-base de exercícios ("cursoCPP", por exemplo), crie um novo diretório para este exercício; por exemplo, "01\_laco\_while".
  - Nesse diretório, crie o arquivo **main.cpp**.
  - Desse modo teremos algo como:  
"<...>/cursoCPP/01\_laco\_while/main.cpp"
- **Execute** o programa. Entre com diferentes valores para testar todos os caminhos do fluxo, conforme os passos abaixo, **comparando cada momento da execução com o respectivo trecho no código fonte**:
  - valor pago **igual** ao valor total da venda;
  - valor pago **maior que** o valor total da venda.
  - valor pago **menor que** valor total da venda;

3.3.2.3 • Laço for

Conforme já vimos, sempre que temos **instruções que precisam ser repetidas** para atingir um objetivo usamos um **laço**. E, em algumas situações, precisamos de um laço capaz de estabelecer, com clareza, uma **progressão**.

Imagine que quiséssemos **somar todos os números** entre um número **inicial** e um **final**. Por exemplo, somar todos os números entre **1** e **4**. Caso não usássemos um laço, teríamos uma escrita absurda que precisaria ser repetida para outros intervalos, indefinidamente.

Por Exemplo:

```
int num = 1 ;    // "num" armazena 1
num = num + 2 ; // agora "num" armazena 3
num = num + 3 ; // agora "num" armazena 6
num = num + 4 ; // agora "num" armazena 10
```

E, se ao invés de 4 o número final fosse 100? Nesse caso teríamos **100 linhas de instrução** ao invés das quatro acima.

Imagine também que quiséssemos calcular o **fatorial de um número**, por exemplo o número **4**. Teríamos o mesmo problema, pois este também é um cálculo **repetitivo**: o número é multiplicado por **ele menos um** até atingir **1**.

```
int num = 4 ;           // "num" armazenada 4
num = num * 3 ;        // agora "num" armazenada 12
num = num * 2 ;        // agora "num" armazenada 24
// num = num * 1; // desnecessário; valor permanecerá o mesmo.
```

Do mesmo modo que no exemplo da soma, se **ao invés de 4** o número fosse **100**, teríamos **99 linhas de instrução ao invés das três acima**.

Laços existem para resolver situações desse tipo, onde uma instrução precisa ser **repetida "n" vezes**. E o laço **"for"** permitirá estabelecer essa repetição **de modo progressivo** (positivo ou negativo), até que um limite seja atingido. Nesses casos, os outros laços também poderiam fazer o mesmo. Mas não com a mesma **clareza**.

O laço **"for"** **repete** as instruções a ele associadas, **até que uma condição torne-se falsa** (e então o laço é **interrompido**), de acordo com a seguinte regra de sintaxe:

```
for ( <início> ; <condição de continuidade> ; <progressão> )
{
    // uma ou mais instruções a executar se a <condição> for verdadeira
}
```

- Se, em um laço **for**, precisamos que mais do que uma instrução seja executada (um bloco de instruções), as instruções devem estar entre **chaves, como acima**.
- Já se existir **apenas uma** instrução, as chaves podem ser **omitidas**:

```
for ( <início> ; <condição de continuidade> ; <progressão> )
    // única instrução a executar se a <condição> for verdadeira
```

### Exemplo 1. Somando números entre um número inicial e um final:



OBS.: neste caso poderíamos resolver esse problema com um **simples cálculo: a soma dos termos de uma progressão aritmética**:

```
int inicial = 1 , final = 100 ; // Intervalo. Substituir valores para teste.
int razao = 1 ; // Distância entre números. Substituir o valor para teste.
int n = (final - inicial + razao) / razao ; // Total de números no intervalo.
int a_n = inicial + ( (n-1)*razao ) ; // Último termo.
int result = ( (inicial + a_n) * n ) / 2 ; // Resultado da soma.
```

Contudo, queremos exercitar o laço **"for"**. Então, a soma seria feita assim:

```
int inicial = 1 , final = 100 , razao = 1 , result ;
// ( <início> ; <condição continuidade> ; <progressão> )
for ( result = 0 ; inicial <= final ; inicial = inicial + razao )
    result = result + inicial ; // Instrução a executar se condição verdadeira
std::cout << "resultado da soma: " << result << "\n";
// Imprime: "resultado da soma: 5050"
```

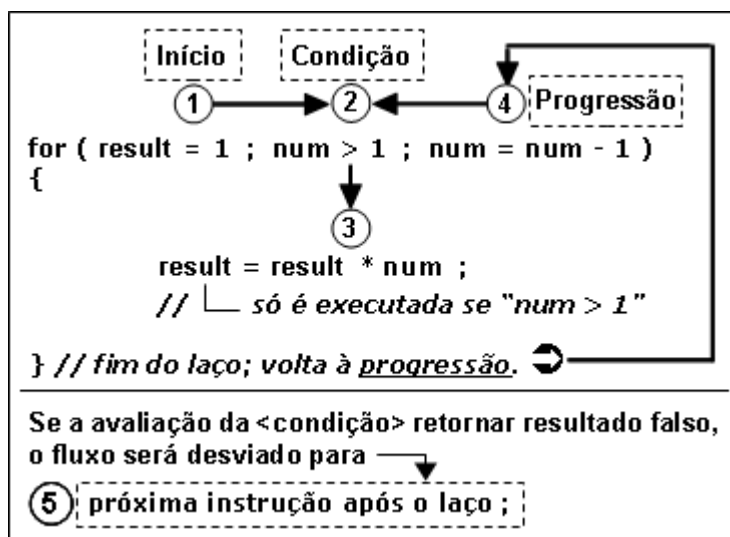
**Exemplo 2. Cálculo de fatorial:**

```
int num = 10 , result ;
for ( result = 1 ; num > 1 ; num = num - 1 )
    result = result * num ; // Instrução a executar se condição verdadeira
std::cout << "resultado para fatorial de 10 : " << result << '\n';
// Imprime: "resultado para fatorial de 10 : 3628800"
```

**Como funciona:**

- A <condição de continuidade> (o **segundo segmento**, após o primeiro ponto e vírgula) é avaliada em **todas** as vezes. Assim, apenas quando "**num**" for **maior-que 1**, a instrução associada ao laço **for** será **executada repetidamente**; se essa condição for falsa, o laço será interrompido (**ou nem iniciará**).
- O <início> (**primeiro segmento**, anterior ao primeiro ponto e vírgula, é executado apenas na **primeira vez**. Em seguida, a **condição** é avaliada.
- A <progressão> (**terceiro segmento**, após o segundo ponto e vírgula) é executado **após** a primeira vez (ao invés de executar <início>); ou seja, é executado na primeira **repetição**. Em seguida, a **condição** é avaliada.

Esse **fluxo** estabelecido pelo "**for**" pode ser visualizado assim:



Observar também os seguintes exemplos:

**a. while: condição inicialmente falsa.**

```
int num = 1;
while ( num < 1 ) // se 'num' menor-que 1
    std::cout << "imprimindo\n";
```

Como **resultado** nada será impresso, pois condição é **falsa** já na **primeira avaliação da condição: "num<1"** (sendo que o valor inicial de 'num' é 1).

**b. for: condição inicialmente falsa.**



```
int num ;
for ( num=1 ; num < 1 ; num=num+1 ) // se 'num' menor-que 1
    std::cout << num << " , ";
std::cout << "\nvalor de 'num' depois do 'for' = " << num << "\n";
```

Como **resultado** da execução, será impresso no monitor de vídeo:

valor de 'num' depois do 'for' = 1

- Nenhum número foi impresso, pois a condição é **falsa** já na **primeira avaliação da condição**: "**num<1**" (sendo que o valor inicial de 'num' é **1**).
- E a mensagem final apenas exibe o valor inicial de "num": **1**.

#### C. **for**: condição inicialmente verdadeira.

```
int num ;
for ( num=1 ; num <= 10 ; num=num+1 ) // 'num' menor ou igual a 10
    std::cout << num << " , ";
std::cout << "\nvalor de 'num' depois do 'for' = " << num << "\n";
```

Como **resultado** da execução, será impresso no monitor de vídeo:

1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10  
valor de 'num' depois do 'for' = 11

- Foram impressos 10 números (de 1 a 10), já que a condição permite a repetição da instrução de impressão **se** "**num<=10**".
- E, após a impressão do número 10, ocorreu a última execução da **progressão** ("**num = num + 1**"), de modo que o valor de "num" tornou-se **11**.
- Em seguida, houve uma última **avaliação da condição**: e o resultado foi **falso**, pois agora 'num' continha **11**. Logo é maior e **não menor ou igual** a 10.
- Assim, **após o laço** é impresso **11**, último valor que foi atribuído a 'num'.

### 3.3.3 • O tipo bool e o resultado de avaliações.

O tipo **bool** não existe em **C**, apenas em **C++**. Acima, quando examinamos os conceitos de verdadeiro e falso, dissemos que se a avaliação de uma condição tem um resultado **falso**, teremos como valor de retorno o inteiro **zero**. Se **verdadeira**, o retorno será o inteiro **um**. Em **C** esse já é, implicitamente, um resultado lógico (ou seja **booleano**).

Em **C++** isso é materialmente válido, mas **formalmente** o retorno de uma avaliação não é do tipo **int** e sim do tipo **bool**: valores **true** ou **false**.

E estas são duas palavras reservadas de **C++**, que também não existem em **C**. Ainda que **true**, na memória, seja armazenado como **um**, e **false** como **zero**, essas duas palavras visam expressar, com maior clareza, os conceitos *booleanos* de verdadeiro e falso.

## 3.4 • Comentários

Este é um assunto sintaticamente trivial. Nada tem a ver com operações e instruções.

Contudo é bom falar nisso desde o início, pois comentários são essenciais na escrita de um código fonte **legível**. Já falamos parcialmente sobre isso.

Vejamos abaixo o que falta dizer sobre comentários.

**Temos dois modos para escrever comentários, com os símbolos:**

- `/* <comentário> */` - válido em **C** e **C++**.
- `// <comentário> <quebra de linha do texto>` - válido em **C++** e **C99**; muitos compiladores, dependendo das opções de compilação o aceitam em **C89**.


**Exemplos:**

```
/* ← inicia um comentário
.....
finaliza um comentário → */
```


Já o símbolo `//` inicia um comentário que é encerrado ao final da linha de texto (*new-line*):

```
// ← comentário encerrado ao final da linha de texto → [quebra de linha]
```

---

 Comentários são importantes para um melhor entendimento do código, explicando o **sentido** de certas operações e rotinas, o qual **nem sempre pode ser deduzido rapidamente do código em si**.

---

 Mas isso **não significa que eles sirvam para tornar legível um código confuso**.

---

**Por exemplo:**

```
double vp = 100 ; // "vp", quem diria, significa "valor do produto" 🖱
```

Ora, por que não escrever **claramente**, tornando o código **auto-explicativo**? Assim:

```
double valor_produto = 100 ; // em princípio, nada a comentar aqui 👍
```

Comentários, usados nos **lugares realmente necessários**, são essenciais, pois o código não é escrito apenas para ser compilado (e o compilador **despreza** comentários), mas também para ser **lido e relido**: quase sempre um código precisa ser melhorado ou até corrigido.

Logo, o código deve ser escrito da maneira mais **legível** (clareza!) que seja possível e deve ter comentários sempre que necessário. Um programa que esteja apresentando erros de execução mas que esteja bem escrito é melhor (porque é mais fácil consertá-lo) do que um programa que esteja funcionando mas esteja mal escrito (porque um dia será preciso alterá-lo, e perderemos muito tempo tentando entender o que ele faz).

## 3.5 • Conclusão

Observando o código de todos exemplos já expostos, no capítulo anterior e neste, podemos inferir algumas das **regras e diretivas** comuns às linguagens **C** e **C++**, isto é, as definições dessas linguagens em relação ao **uso da memória, operações e fluxo de processamento**, inclusive **laços**.

Nos próximos dois capítulos, veremos com mais detalhes a definição e uso de cada um desses elementos nessas linguagens:

- **memória**: capítulo **4**, página **97**.
- **fluxo de processamento**: capítulo **5**, página **131**.

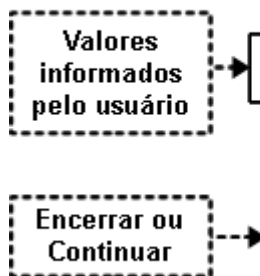
## 3.6 • Revisão do Capítulo 3

### 3.6.1 • Exercício.

Tente resolver o exercício abaixo. Em seguida, compare sua solução com a que está no **Anexo B-3-1, página 431**. Caso não entenda, fale com o instrutor.

**Enunciado:** implemente a soma dos números entre um número **inicial** e um **final**, com a distância "**razão**" entre cada número. Esses 3 valores devem ser informados. E o usuário deverá decidir também se deseja um novo cálculo ou encerrar o programa.

**Resultado que deve ser impresso:**



```

C:\> Prompt de comando

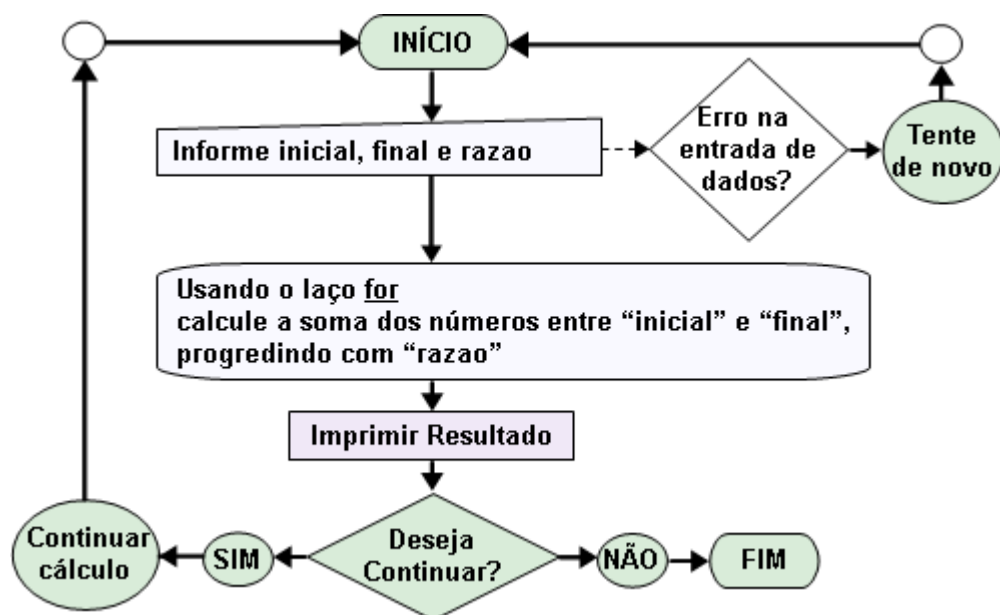
informe os numeros inicial, final e a razao
- nessa ordem:
10
20
2
resultado da soma: 90

deseja realizar novo calculo?
<0 para encerrar,
qualquer outro numero para continuar>
0

fim de processamento
```

- Para a soma, use o **laço for** e não o cálculo de soma dos termos da progressão, pois aqui queremos justamente **exercitar o laço**.
- Os números "inicial", "final" e a distância entre cada número ("razão") deverão ser **informados pelo usuário** (use **std::cin**).
- O cálculo deverá ser **repetido** para outros números, até que o usuário decida interromper o processamento.

O objetivo a atingir pode ser visualizado no seguinte fluxo:



**Modelo** para implementação.

Esse fluxo pode ser implementado de diversas maneiras.

**Por exemplo, de acordo com o modelo** exibido ao lado:

```

while ( < ...> )
{
    // a) entrada de dados
    if ( <erro_na_entrada> )
        // imprime mensagem de erro
    else
    {
        for ( < ...> ; < ...> ; <...> )
        {
            // b) soma...
        }

        // c) imprime resultado
        // d) pergunta ao usuário: deseja continuar?
        // e) a entrada do usuário pode modificar a condição
        // do laço "while", interrompendo o laço.
    } // fim do if
} // fim do while
  
```

**Para atingir o objetivo, siga estes passos:**

- Usando um editor ou um ambiente, crie um novo diretório para este exercício; por exemplo, "02\_lacos\_while\_for", dentro do diretório-base "cursoCPP".

- b. Crie um arquivo e o salve, por exemplo, com o nome **"main.cpp"**.

Desse modo teremos algo como:

**"<...>/cursoCPP/02\_lacos\_while\_for/main.cpp"**

- c. Comece a escrever o código na função **main**. O programa deve pedir ao usuário que informe:

- Os números inicial e final para a soma.
- A razão (distância ou valor a somar para atingir o próximo número da série).
  - Neste exercício, a razão **não pode ser zero** (analise o que foi informado).
- Caso o **inicial seja maior que o final**, você poderia prosseguir se a razão fosse **negativa**. Mas seria preciso também **alterar a condição** do laço **for**. Para os fins deste exercício, você pode seguir um caminho mais simples:
  - **se inicial for maior que o final**, troque os valores de inicial e final (uma operação de *swap*) de modo que a progressão continue sendo positiva (progredindo do menor para o maior).
    - Para trocar o valor de dois números, você pode escrever o código necessário para um *swap* (troca), ou, simplesmente, usar:

```
if ( inicial > final )
    std::swap( inicial, final ) ;
```

- Uma vez garantido que o **inicial é menor que o final**, a **razão** deve ser um número **positivo** (analise o que foi informado). Caso seja negativo, você pode inverter o sinal assim:

```
if ( razao < 0 )
    razao = -razao ;
```

- d. Implemente o laço **for**: se tiver dificuldades, olhe o **exemplo da soma de números** (página **87**, acima).

- e. Após o laço, **imprima o resultado** da soma.

- f. Agora, **volte ao início**, para que o usuário continue informando novos valores para cálculo.

- Dica: isso pode ser resolvido com um laço **while**. Nesse caso, o laço **for** estará dentro do laço **while** ("aninhado").
- Estabeleça uma **condição de fim** para o processamento. Por exemplo, após concluir um cálculo e imprimir o resultado, pergunte ao usuário "deseja continuar?".
- Tente usar o **while sem** usar o **break** para interromper o laço, ao contrário do que foi feito no exemplo da "operação de vendas". Se não conseguir, pelo menos tente fazer o mesmo que foi feito nesse exemplo (página **82**, acima).

- **Informação importante:** como a **entrada de dados** estará dentro de um laço, e assim ela também será repetida, caso ocorra erro (se o usuário digitar uma entrada inválida) a próxima operação de entrada de dados irá falhar. Então é preciso fazer o seguinte:

```
while ( <...> )
{
    // a) entrada de dados
    for ( <...> ; <...> ; <...> )
    {
        // b) soma...
    }
    // c) imprime resultado
    // d) pergunta ao usuário:
    // deseja continuar?
}
```

```
// solicita ao usuário os 3 números necessários:
int inicial, final, razao, result ;
std::cout << "\ninforme os números inicial, final e a razão\n";
std::cin >> inicial >> final >> razao ;

// se houve falha na entrada:
if ( std::cin.fail() )
{
    std::cout << "valores incorretos: tente outra vez\n";
    // como houve falha é preciso limpar os flags de erro de "cin":
    std::cin.clear( ); ☹

    // e desprezar quebras de linha pendentes no buffer de "cin":
    std::cin.ignore(std::numeric_limits<int>::max(), '\n') ; ☹
}
```

- As funções "**clear**" e "**ignore**" fazem coisas diferentes, mas o que interessa aqui é que, com elas, o estado de "**cin**" é restaurado para a situação anterior aos erros. Veremos os detalhes disso mais a frente.
- Observe que o segundo argumento passado para "**ignore**" é o caractere '**\n**', entre **aspas simples** (e não duplas). Veremos na seção **4.10.5**, página **123**, a diferença entre aspas simples e duplas para caracteres.
- Dependendo do compilador, será necessário incluir o arquivo onde "**numeric\_limits**" está declarado. Então, insira no topo do arquivo **main.cpp**:

```
#include <iostream>
#include <limits> ☹
```

**g. Compilar** (usando ambientes de desenvolvimento ou a linha de comando).

- Se ocorreram **erros de compilação**, verifique e corrija as linhas de erro.
  - Dúvidas? Reveja as dicas sobre identificação de erros (seção **2.4**, página **54**, acima). E as regras **básicas de gramática** (seção **2.2.4**, página **46**, acima).

**h. Execute** o programa. Observe se funciona corretamente.



**Compare o que você fez com a solução da apostila: página 431.**

### 3.6.2 • Questões para revisão

Responda às questões abaixo, assinalando **todas** as respostas corretas (**uma ou mais**). Em seguida, compare suas respostas com as respostas localizadas no **Anexo B-3-2**, página **433**. Caso não entenda, encaminhe as dúvidas ao instrutor.

☹ Nas questões abaixo, considere as seguintes declarações iniciais:  
**int x , y ;**

**Cap.3 - 1.** A respeito de uso da **memória**, assinale as afirmações corretas

- a. ☐ Não é preciso ter preocupações especiais com a memória de um computador. Isso é resolvido pelo compilador.
- b. ☐ Uma memória deve ser reservada e qualificada com um determinado **tipo** pelo programador. Por exemplo, usando **int**, como na expressão `[ int x ; ]`
- c. ☐ A expressão `[ x = 10 ; ]` visa **confirmar** se '**x**' contem o valor **10**.
- d. ☐ A expressão `[ x = 10 ; ]` visa **armazenar** o valor **10** em '**x**', copiando esse valor para a área de memória à qual associamos o nome '**x**'.

**Cap.3 - 2.** Sobre **fluxo de processamento**, podemos dizer que:

- a. ☐ O processamento é **sempre** baseado na execução sequencial de um certo número de instruções.
- b. ☐ O processamento pode ser bifurcado em caminhos diferentes, através de **tomadas de decisão**.
- c. ☐ A expressão `[ if ( x > y ) ]` contem uma operação que leva à seguinte avaliação: "será que '**x**' pode ser **armazenado** em '**y**' ?
- d. ☐ Em `[ if ( x > y ) ]` temos uma operação que implica nesta avaliação: "o valor armazenado em '**x**' é **maior-que** o valor armazenado em '**y**' ?
- e. ☐ Após `[ if ( x > y ) ]` o processamento poderá seguir um caminho diferente se o resultado da avaliação dessa condição for verdadeiro.

**Cap.3 - 3.** Sobre os conceitos de **verdadeiro** e **falso**, podemos dizer que:

- a. ☐ A avaliação da expressão contida em `[ if ( x > y ) ]` retornará um resultado **verdadeiro** se '**x**' for **maior-que** '**y**'; do contrário retornará um resultado **falso**.
- b. ☐ A avaliação da expressão contida em `[ if ( x >= y ) ]` retornará um resultado **verdadeiro** se '**x**' for **maior-que** '**y**'; do contrário retornará um resultado **falso**.
- c. ☐ A expressão contida em `[ if ( x ) ]` **não é aceita** pelas linguagens **C** e **C++**. Logo, ocorrerá um erro de compilação.
- d. ☐ A avaliação da expressão contida em `[ if ( x ) ]` retornará um resultado **falso** se o valor de '**x**' for igual a **0**(zero); e retornará um resultado **verdadeiro** se o valor de '**x**' for **exatamente** igual a **1**(um).
- e. ☐ A avaliação da expressão contida em `[ if ( x ) ]` retornará um resultado **falso** se o valor de '**x**' for igual a **0**(zero); e retornará um resultado **verdadeiro** se o valor de '**x**' for **diferente** de **0**(zero)

**Cap.3 - 4.** O que será impresso na execução do código abaixo?

```
x = 5 ;  
y = x + 1 ;  
x = x + 2 ;  
if ( x >= y )  
    std::cout << "x é maior ou igual a y\n" ;  
else  
    std::cout << "x é menor que y\n" ;
```

- 
- a. ☐ Será impresso **x é maior ou igual a y**
- 
- b. ☐ Será impresso **x é menor que y**
- 
- c. ☐ **Nada** será impresso.
- 
- d. ☐ Será impresso **x é maior ou igual a y** e, **em seguida**, será impresso **x é menor que y**
- 

**Cap.3 - 5.** Sobre **laços**, podemos dizer que:

- 
- a. ☐ Um laço **sempre** repete, **infinitamente**, um certo número de instruções.
- 
- b. ☐ Um laço pode conter uma avaliação de condição que, caso seja ou torne-se falsa, resultará na interrupção da repetição das instruções a ele associadas.
- 
- c. ☐ O laço **while** contém uma avaliação de condição que permite interrompê-lo. Já o laço **for** contém apenas uma possibilidade de **progressão**, mas não prevê uma avaliação de condição.
- 

**Cap.3 - 6.** Quantas vezes serão executadas as instruções associadas ao seguinte laço:

```
for ( x = 1 ; x < 1 ; x = x + 1 )  
{  
    // instruções...  
}
```

- 
- a. ☐ **Uma** vez.
- 
- b. ☐ **Duas** vezes.
- 
- c. ☐ **Nunca** serão executadas.
- 
- d. ☐ Serão executadas **infinitamente**.
- 



**Compare o que você fez com a solução da apostila: página 433**



## • Capítulo 4

### ▪ Memória: tipos, declarações e definições

Neste capítulo iremos aprofundar alguns dos tópicos vistos nos capítulos anteriores, acrescentando novos elementos.

Veremos detalhes sobre o uso da memória: valores, tipos de dados, objetos, variáveis, constantes e suas declarações.

Teremos também uma abordagem inicial sobre o tratamento de dados armazenados em série: vetores.

4.1 • Tópicos de destaque neste capítulo.....	99
4.2 • Reserva e acesso de memória em C e C++ .....	100
4.2.1 • Tipos de dados.....	100
4.2.2 • Escolhendo o tipo adequado.....	100
4.2.3 • Porque reservar memórias com um tipo.....	102
4.2.4 • Para lembrar.....	102
4.2.5 • Criando sinônimos de tipos (C++98 e C++11).....	103
4.2.5.1 • Para que servem sinônimos de tipos.....	103
4.3 • Inicialização e atribuição.....	103
4.4 • Variáveis e constantes.....	104
4.4.1 • Porque associar constantes a um nome.....	104
4.4.2 • Declarando constantes com <i>enum</i> .....	105
4.4.3 • Enum: diferenças entre C e C++ e entre C++98 e C++11.....	106
4.4.3.1 • Denominação de um tipo criado com enum.....	106
4.4.3.2 • Enum sem e com escopo próprio.....	107
4.4.3.3 • Definindo o tipo inteiro subjacente de um enum (C++11).....	107
4.4.3.4 • Exemplos de escopo e especificação do seu tipo inteiro.....	107
4.4.4 • Constantes nomeadas em um único lugar.....	108
4.5 • Declarações auto (C++11).....	109
4.6 • Conceitos: valores, tipos e objetos.....	109
4.7 • Conversões entre tipos.....	110
4.8 • Variáveis classificadas como <i>static</i> e <i>extern</i> .....	113
4.8.1 • Variáveis <i>static</i> de um bloco.....	113
4.8.2 • Variáveis <i>static</i> de um módulo e variáveis <i>extern</i> .....	114
4.8.3 • Conformidade com C++.....	115
4.9 • Como a memória é organizada.....	115
4.9.1 • Memória global ou estática.....	115
4.9.2 • A pilha ( <i>stack</i> ).....	115
4.9.3 • Livre alocação.....	115
4.10 • Vetores.....	116
4.10.1 • Vetores: o que são e para que servem.....	116
4.10.2 • <i>std::string</i> e <i>std::vector</i> .....	116
4.10.3 • Vetores na linguagem C.....	119
4.10.4 • Usos seguros de índices em vetores.....	122
4.10.5 • Caracteres: aspas simples e duplas.....	123
4.11 • Revisão do Capítulo 4.....	124
4.11.1 • Exercício.....	124
4.11.2 • Questões para revisão.....	128

4.1 • Tópicos de destaque neste capítulo.....	99
4.2 • Reserva e acesso de memória em C e C++.....	100
4.2.1 • Tipos de dados.....	100
4.2.2 • Escolhendo o tipo adequado.....	100
4.2.3 • Porque reservar memórias com um tipo.....	102
4.2.4 • Para lembrar.....	102
4.2.5 • Criando sinônimos de tipos (C++98 e C++11).....	103
4.2.5.1 • Para que servem sinônimos de tipos.....	103
4.3 • Inicialização e atribuição.....	103
4.4 • Variáveis e constantes.....	104
4.4.1 • Porque associar constantes a um nome.....	104
4.4.2 • Declarando constantes com <i>enum</i> .....	105
4.4.3 • Enum: diferenças entre C e C++ e entre C++98 e C++11.....	106
4.4.3.1 • Denominação de um tipo criado com enum.....	106
4.4.3.2 • Enum sem e com escopo próprio.....	107
4.4.3.3 • Definindo o tipo inteiro subjacente de um enum (C++11).....	107
4.4.3.4 • Exemplos de escopo e especificação do seu tipo inteiro.....	107
4.4.4 • Constantes nomeadas em um único lugar.....	108
4.5 • Declarações auto (C++11).....	109
4.6 • Conceitos: valores, tipos e objetos.....	109
4.7 • Conversões entre tipos.....	110
4.8 • Variáveis classificadas como <i>static</i> e <i>extern</i> .....	113
4.8.1 • Variáveis <i>static</i> de um bloco.....	113
4.8.2 • Variáveis <i>static</i> de um módulo e variáveis <i>extern</i> .....	114
4.8.3 • Conformidade com C++.....	115
4.9 • Como a memória é organizada.....	115
4.9.1 • Memória global ou estática.....	115
4.9.2 • A pilha ( <i>stack</i> ).....	115
4.9.3 • Livre alocação.....	115
4.10 • Vetores.....	116
4.10.1 • Vetores: o que são e para que servem.....	116
4.10.2 • <i>std::string</i> e <i>std::vector</i> .....	116
4.10.3 • Vetores na linguagem C.....	119
4.10.4 • Usos seguros de índices em vetores.....	122
4.10.5 • Caracteres: aspas simples e duplas.....	123
4.11 • Revisão do Capítulo 4.....	124
4.11.1 • Exercício.....	124
4.11.2 • Questões para revisão.....	128

## 4.1 • Tópicos de destaque neste capítulo

### ➤ **Revendo os princípios básicos de programação e detalhando sua implementação em C e C++: memória.**

- ♦ **Memória:** reservar áreas da memória para que possam ser acessadas nas operações a executar.
- ♦ Ao reservar uma área de memória em C e C++, devemos indicar qual é o seu tipo.
- ♦ Um tipo indica o tamanho (quantidade de bytes) da área de memória reservada e a forma como os dados serão nela armazenados (forma de armazenamento).
- ♦ Há tipos diferentes para números inteiros e números reais.
- ♦ Devido à forma de armazenamento, números reais têm um acesso mais lento do que números inteiros.
- ♦ Declaração de uma memória. Diferença entre inicialização e atribuição.
- ♦ É possível fazer conversões entre determinados tipos: algumas são seguras e outras não.
- ♦ Convenções para tipos que permitem expressar caracteres.
- ♦ Sinônimos para tipos.

### ➤ **Memórias: Variáveis e Constantes.**

- ♦ Variáveis podem (ou não) ser inicializadas e, depois, sofrer sucessivas atribuições.
- ♦ Constantes podem apenas ser inicializadas; não podem receber atribuições.
- ♦ Porque associar constantes a um nome.
- ♦ Declarando constantes.
- ♦ Declarando constantes enumeradas.
- ♦ Declarações *auto* (C++11)

### ➤ **Vetores.**

- ♦ Armazenando séries de valores de um mesmo tipo.
- ♦ Trabalhando com vetores de caracteres: std::string.
- ♦ Trabalhando com vetores de qualquer tipo que aceite a operação de atribuição: std::vector.
- ♦ Vetores no estilo "C": usos e problemas.
- ♦ Usos seguros de índices em vetores.

## 4.2 • Reserva e acesso de memória em C e C++

### 4.2.1 • Tipos de dados

Observe que, nos exemplos acima, é utilizada a palavra “**int**”.

```
int valor_produto ;
```

```
int valor_imposto ;
```

**int** é uma palavra reservada de C e C++ que indica como será feita uma determinada **reserva de memória**. Ele indica um **tipo de dados**. E a linha onde reservamos uma memória (denominada **declaração**) deve, obrigatoriamente, indicar um **tipo** e um **identificador**(nome) para essa memória.

Nesse caso, ao usar o **int**, estamos dizendo que precisamos da reserva de uma memória cujo **tipo** seja capaz de armazenar **números inteiros** com um **determinado tamanho**.

Podemos também declarar memórias **do mesmo tipo em uma única linha**, separando os seus identificadores (nomes) com **vírgulas**:

```
int valor_produto, valor_imposto;
```

E, em seguida, **atribuir valores** a qualquer uma dessas memórias reservadas:

```
valor_produto = 100 ; // o valor 100 foi copiado para "valor_produto"
```

No diagrama simplificado exposto abaixo, visualizamos o estado da memória após a alteração de “**valor\_produto**”:

<i>Apelidos</i> (símbolos) para identificar, no código fonte, cada <b>posição de memória</b> reservada ( <i>esses nomes não existirão no executável</i> )	“valor_produto”	“valor_imposto”
Posição na memória ou “endereço de memória” (por hipótese)	1000	1004
Valor armazenado	100	? (“lixo”)

Veremos ainda neste capítulo (seção **4.6**, página **109**) que um **tipo qualifica** uma área de memória, permitindo representá-la como um **objeto** (um objeto é um valor variável ou constante de determinado tipo).


### 4.2.2 • Escolhendo o tipo adequado

Podemos observar também que, para o exemplo acima, o tipo **int** **não parece muito adequado**. Pois com ele não podemos armazenar números que suportem uma **parte fracionária**.


E, se estamos falando em “*valor de um produto*”, precisamos prever a ocorrência de **centavos**, havendo assim a necessidade de casas decimais. Pois, ainda que no exemplo não tenham sido usados os centavos, eles poderão aparecer a qualquer momento, devido à natureza do valor em questão. Por isso, ao invés de um número inteiro, precisamos de um **número real**, o que, em computação, significa um número com **ponto flutuante** (que é um formato de representação digital de números reais).

Para armazenar números com ponto flutuante, temos **outros tipos**, diferentes do **int**, como é o caso do **tipo double**.

Desse modo, o código do nosso exemplo poderia ser escrito assim:

```
double valor_produto = 100 ;           //  poderia ter centavos...
double valor_imposto = 10 ;             // idem...
double valor_venda = valor_produto + valor_imposto ; // idem...
double valor_pago = 110 ;               // idem...
```

E qual a razão pela qual algumas linguagens usam **diferentes palavras reservadas para a declaração de números inteiros e números reais?**


 O motivo, além da possível diferença de tamanho, é *performance*: um número **inteiro** pode ser **armazenado e acessado mais rapidamente**, ao passo que um número **real** (ponto flutuante) exige uma forma de armazenamento e acesso **mais complexa e lenta**.

Afinal, não podemos esquecer que computadores são baseados em números de base **binária**, que, por natureza, são **inteiros**. Assim, o **ponto flutuante**, para armazenar a parte fracionária, exige uma forma de armazenamento diferenciada, dividindo o número em uma **mantissa** e um **expoente**, o que torna o acesso **mais lento**.

 Se quiser mais informações sobre o assunto "**ponto flutuante**", eis aqui um **ponto de partida**: [http://pt.wikipedia.org/wiki/Ponto\\_flutuante](http://pt.wikipedia.org/wiki/Ponto_flutuante).

Ficamos sabendo acima que o uso de números reais implica em um acesso mais lento. Mas, se o problema que estamos resolvendo **exige o uso** de números reais (por exemplo, para armazenar os centavos), então temos que usar um **tipo** de número **real**.

**Em conclusão, o princípio básico é:**


 **Reservar memória** em **C** e **C++**, significa **declarar** uma memória com um **determinado tipo**.

E um **tipo** indica uma **forma de armazenamento e um tamanho**.

Os tipos são definidos com **palavras reservadas**, como vemos abaixo, na tabela reduzida de tipos (apenas os que usaremos **inicialmente**):

Alguns tipos:		
Tipo	Tamanho <u>mínimo</u> em bytes	Forma de armazenamento
<b>int</b>	2, 4, 8, ... depende da plataforma	inteiro
<b>double</b>	8	ponto flutuante
<b>float</b>	4	ponto flutuante
<b>enum</b>	Em <b>C++98</b> : entre 1 byte (char) e o tamanho que tenha o <b>int</b> , dependendo do valor usado. Mas, a partir do <b>C++11</b> , deve ser um <b>int</b> .	inteiro (enumera <u>constantes</u> )

Na tabela acima relacionamos **alguns** dos tipos referentes a **valores** (apenas aqueles que usaremos logo abaixo).

 Existem **outros tipos**, como veremos depois. Eles são mostrados na tabela localizada no "**guia de consulta rápida**", seção **1**, página **477**.


### 4.2.3 • Porque reservar memórias com um tipo

Devido à reserva de memória com tipos explicitamente definidos, temos vantagens:

- Um uso mais claro da memória: sempre poderemos saber o que deve e pode ser feito em determinada área de memória.
- E o compilador, por sua vez, poderá analisar se estamos usando essa memória conforme sua especificação, e assim, se tentarmos atribuir-lhe um valor indevido, o compilador poderá emitir uma **mensagem de erro ou de advertência**.
  - Por exemplo: se tentarmos fazer

```
int x ;
x = 5.3 ;    // 5.3 é do tipo double
```

- O compilador emitirá um aviso, pois o valor **5.3** será **truncado** para **5** ao ser copiado para 'x', já que esta memória é do tipo **int** e **não suporta** o armazenamento da parte fracionária.
- Além disso, o compilador poderá resolver totalmente o problema da alocação dessa memória, e nenhum tipo de interpretação ou análise será necessária em tempo de execução para que a alocação seja efetuada.

 Reservar memória com um **tipo** (forma de armazenamento e tamanho próprios) permite **evitar erros** de atribuição, além de **ganhos de performance**.

### 4.2.4 • Para lembrar...

Existem linguagens em que é possível ocupar a memória sem antes declarar para que vamos utilizá-la.

E existem outras (como é o caso de **C** e **C++**) em que é obrigatório que, antes de acessarmos uma região de memória, solicitemos explicitamente uma reserva da quantidade de memória necessária, indicando também o uso que será dado a esse pedaço da memória.

E isso é feito indicando-se um **tipo** para a memória que deve ser reservada. Um tipo deve identificar duas características para a memória que está sendo reservada:

**1) tamanho:** um tipo sempre tem um tamanho fixo e conhecido em tempo de compilação;

**2) forma de armazenamento:** a forma de armazenamento também deve ser conhecida em tempo de compilação. Um número inteiro, por exemplo, é armazenado de um modo diferente do modo com que é armazenado um número real que admite uma parte fracionária.

Dissemos que a linguagem exige o uso de tipos para a alocação de memórias. E é bom ressaltar que isto nos proporciona os seguintes benefícios:

**1) segurança:** se os tipos são conhecidos o compilador pode analisar se estamos usando uma memória reservada de modo compatível com o tipo declarado em sua criação. E caso o uso esteja incorreto o compilador **poderá** emitir mensagens de erro ou aviso, evitando que ocorram erros já em tempo de execução.


**2) eficiência (performance):** se os tamanhos e modos de armazenamento são conhecidos em tempo de compilação, o compilador pode gerar o código necessário para a alocação e uso da memória.

Por isso mesmo, como veremos no capítulo **6**, página 186 ("**Ponteiros e referências**") na linguagem **C++**, quando é preciso alocar quantidades de memória que só serão conhecidas em tempo de execução, é o próprio programador quem deve criar o código necessário para a sua alocação e liberação.

**Mas lembre também que:**

- Em **C**, a declaração de variáveis deve **iniciar a escrita de um bloco de código**, **não** podendo estar após qualquer outra linha de instrução desse bloco.
- Em **C++**, a declaração de variáveis **pode estar após linhas de instrução**; contudo deve sempre preceder a linha de instrução que utilizar **aquela variável**.

## 4.2.5 • Criando sinônimos de tipos (C++98 e C++11)

 O programador pode criar **sinônimos** para tipos de dados já existentes, através da palavra reservada **typedef** (**C**, **C++98**, ...) ou através de **type alias** (**C++11**).

### Sintaxes:

- 1) **typedef** <tipo existente> <sinônimo para esse tipo> ; // em **C**, **C++98**, **11...**  
ou
- 2) **using** <sinônimo para o tipo> = <tipo existente>; // só a partir de **C++11**

A vantagem em se usar o **type alias** (com **using**) é que podemos definir mais facilmente sinônimos para *templates* (veremos **templates e criação de sinônimos para templates**) no capítulo 8, página 264 - "Programação genérica").

### Exemplos:

```
typedef unsigned int  UINT ;
```

Neste exemplo criamos o sinônimo "UINT" para o tipo "unsigned int".

```
typedef unsigned short  USHORT;
```

Neste exemplo criamos o sinônimo "USHORT" para o tipo "unsigned short".

```
using USHORT = unsigned short ;
```

Neste exemplo, também criamos o sinônimo "USHORT" para o tipo "unsigned short".

### 4.2.5.1 • Para que servem sinônimos de tipos

Os sinônimos podem simplificar a declaração de variáveis com nomes de tipo muito longos (como *unsigned int*). Mas também são úteis para **portabilidade**.

#### Por exemplo:

podemos criar o sinônimo **real\_t**. O **real\_t** será um sinônimo para **double**(8 bytes) em plataformas onde o **double** é aceito. Mas será um sinônimo para **float**(4 bytes) em plataformas onde o **double não é aceito** (geralmente, em certos embarcados com memória reduzida).


## 4.3 • Inicialização e atribuição

**Inicialização:** é quando **atribuímos** um valor a uma memória que está sendo reservada na própria linha em que ela é **declarada** (ou seja, no momento em que é reservada):

```
int valor_produto = 100 , valor_imposto; // "valor_produto" foi inicializada
```

Já em **qualquer outra linha** que não seja a da declaração, uma operação de cópia de valor é denominada **atribuição**:

```
valor_imposto = 10 ; // "valor_imposto" recebeu uma atribuição.
```

 Quando atribuímos um valor a uma memória na linha de sua declaração, temos uma operação denominada **inicialização (e não atribuição)**. Isto é: a variável é **declarada** (com um **tipo**) e **definida** (com um **valor**) em uma **mesma** linha.

---

Já uma **atribuição** é a cópia de um valor para uma memória de valor variável em qualquer linha **que não seja a de sua declaração**.

---

No item **abaixo (variáveis e constantes)** veremos uma situação em que isso fará **uma grande diferença**. E, mais tarde, ainda veremos outras.

---

## 4.4 • Variáveis e constantes

Quando reservamos uma memória para armazenar valores, precisamos definir também de que modo esses valores irão ser usados.

- Se essa memória admitir alterações no valor armazenado, então dizemos que ela é uma **variável**. Poderá aceitar **novas atribuições** de valor.
- Mas existem casos em que um determinada memória é **imutável**. Ela é apenas **inicializada** com um valor que **não mais** será alterado. Dizemos então que essa é uma memória **constante**. Não poderá aceitar qualquer atribuição posterior.

### Exemplos:

```
double valor_produto ; // pode ser inicializada aqui ou
                        // receber valores mais tarde; é uma variável.
const int ultimo_mes = 12 ;
👉 // é uma constante: deve ser inicializada aqui obrigatoriamente;
    // e não poderá ser alterada após essa linha.
```

E porque motivos precisamos **declarar constantes**? Não poderíamos simplesmente usar o número “12” diretamente, sempre que precisássemos nos referir ao último mês de um ano? **Exemplo:**

```
int mes;
// ...
if ( mes == 12 ) // se "mes" for igual a 12...
    // ... pagar o décimo-terceiro salário.
```

Neste caso, o uso direto do número constante “12”, parece claro: afinal estamos comparando esse número com uma variável denominada “mes”. Logo, esse número **parece** referir-se ao último mês de um ano. Mas **nem sempre** um valor usado diretamente aparenta o seu significado.

### 4.4.1 • Porque associar constantes a um nome

#### Exemplo:

```
int x;
x = 80 ; // 🤔 o que significa "80" aqui ? Mistério...
```

Se, ao invés de usar constantes diretamente, associarmos o valor a um **nome**, o código ficará muito mais **claro** e fácil de ler/entender.

```
int x;
const int screen_max_x = 80 ; // o nome indica que essa é a maior
// ...                        // coordenada "x" da tela
x = screen_max_x ; // 👍 OK: nenhuma dúvida sobre o que está sendo feito.
```

---

🔗 Para que o código seja o mais **claro** possível, adote, como **regra geral**, sempre associar valores constantes a um **nome**, mesmo em casos em que uma constante **pareça** clara em um determinado **contexto** (mas pode ficar obscura em um outro). O código fonte deve ser sempre **auto-explicativo**.

---



Portanto, a comparação feita acima entre "mes" e "12" ficaria **melhor** assim:

```
const int ultimo_mes = 12 ;
int mes ; // ...
if ( mes == ultimo_mes ) // nada a explicar ou a comentar...
    // ... pagar o décimo-terceiro salário.
```

Neste caso do "último mês", talvez fosse útil associar **cada mês do ano** a um nome de constante, pois muitas vezes precisamos nos referir a um mês qualquer e não apenas ao último mês do ano.

**Exemplo:**

```
int mes ; // ...
if ( mes == 9 ) // se o "mes" for setembro...
    // ... marcar o dia 7 como feriado.
```

#### 4.4.2 • Declarando constantes com *enum*

Quando temos um determinado **conjunto de constantes** que formam uma **sequência** (como é o caso dos **meses**), podemos usar uma outra forma mais prática e segura de declarar constantes, através do tipo **enum**.

```
enum { const_0 , const_1 }; // const_0 tem o valor 0; const_1 tem o valor 1
```

Ele é um **enumerador**, que permite criar uma **lista de constantes**, separadas por vírgulas, onde:

- o valor de cada uma, por *default*, é sempre o valor da constante **anterior mais um**, sendo que o valor da **primeira** delas, por *default*, é **zero**;
- **exceto se forcarmos** alguma delas para um valor específico - e as seguintes continuarão seguindo a regra "**anterior mais um**", a não ser que também tenham seus valores especificados explicitamente.

**Exemplo:**

```
enum { Janeiro=1 , Fevereiro , Marco , Abril , Maio , Junho , Julho ,
      Agosto , Setembro , Outubro , Novembro , Dezembro } ;
```

Na lista de constantes acima "**Janeiro**" teve seu valor forçado para **1** (do contrário seria **zero**), e as demais seguem a regra "**anterior mais um**". Desse modo, "**Fevereiro**" tem o valor **2** - e assim sucessivamente até "**Dezembro**" que terá o valor **12**.

**Continuando o exemplo:**

```
int mes; // ...

if ( mes == Dezembro ) // nada a explicar ou a comentar... 👍
    // ... pagar o décimo-terceiro salário.

if ( mes == Setembro ) // nada a explicar ou a comentar... 👍
    // ... marcar o dia 7 como feriado.
```

Há uma forma ainda melhor de usar o **enum**, pois ele permite que criemos também um **nome de tipo** associado à **lista de constantes**.

Desse modo, poderemos declarar **variáveis desse tipo**, e com isso, essas variáveis serão amarradas à lista de constantes, de forma que só aceitarão valores presentes nessa lista (ou teremos um erro de compilação).

Pois quando fazemos "**int mes**", declaramos uma variável que está sendo usada para armazenar um dos valores da lista enumerada. Mas, sendo "**int**", poderá aceitar qualquer outro valor inteiro, o que não é o nosso objetivo. Melhor seria então se criássemos **um tipo que impedisse qualquer atribuição aleatória**.

**Exemplo:**

```
enum Meses { Janeiro=1, Fevereiro, Marco, Abril, Maio, Junho, Julho,
             Agosto, Setembro, Outubro, Novembro, Dezembro } ;
```





Criamos o **tipo "Meses"** amarrado à **lista de constantes**. Desse modo, variáveis declaradas com esse tipo **só aceitarão valores da sua lista**.

**Continuando o exemplo:**

```
Meses mes; // Declarei uma variável do tipo "Meses".
// ...

if ( mes == Dezembro ) // nada a explicar ou a comentar...
    // ... pagar o décimo-terceiro salário.

if ( mes == Setembro ) // nada a explicar ou a comentar...
    // ... marcar o dia 7 como feriado.


mes = 31 ; //  Erro de compilação. OK: isso é bom... faça a correção.
mes = 10 ; //  Erro de compilação. Idem...
```



Apesar do valor **10** coincidir com o valor da constante "**Outubro**" da lista, temos um erro, pois a constante **10**, por *default*, é do tipo **int**; e variáveis do tipo "**Meses**" **só aceitam** as constantes nomeadas na **sua lista**.



Na **Linguagem C** isso **não seria considerado um erro e não seria acusado pelo compilador**. Se você escreveu coisas assim em **C**, **corrija** para que possa compilar em **C++**.

```
mes = Outubro ; //  correto.
int x = Outubro ; // também correto pois "Meses" pode ser convertido implicitamente
                  // para int. Nesse caso, o valor de 'x' será 10.
```

**4.4.3 ▪ Enum: diferenças entre C e C++ e entre C++98 e C++11**

Já vimos acima (no exemplo "enum Meses") que em **C** podemos atribuir um valor inteiro qualquer a uma variável do **tipo Meses**. Felizmente, **C++ não manteve a compatibilidade** com **C** neste aspecto. Tal variável só aceita um valor da lista do seu *enum*.

**4.4.3.1 ▪ Denominação de um tipo criado com enum.**

Em **C++**, o nome de um tipo criado através de **enum** é diretamente o nome designado pelo programador.

Assim em

```
enum Semana { Domingo, Segunda, Terca, Quarta, Quinta, Sexta, Sabado } ;
```

O **nome do tipo** é **Semana**. Desse modo declaramos variáveis como abaixo:

```
Semana Variavel ;
```

Mas, como **C** considera a própria palavra reservada **enum** como **parte** do nome do tipo, em **C++** é admitido também (por compatibilidade) que em declarações de variáveis seja usado o nome designado pelo programador, precedido da palavra reservada "**enum**", conforme exemplificado abaixo:

```
enum Semana Variavel ; // em C (mas não em C++) isto seria obrigatório.
```

Por isso, em **C**, é muito comum usar **sinônimos**:

```
typedef enum Semana Semana; // desnecessário em C++
Semana Variavel;
```

### 4.4.3.2 • Enum sem e com escopo próprio

1) Em C++98 temos um problema: o enum **não** tem **escopo** próprio. Por isso se fizermos:

```
enum Semana { Domingo, Segunda, Terca, Quarta, Quinta, Sexta, Sabado } ;
// e, no mesmo escopo, declararmos:
// int Domingo = 5; // ERRO: redefinição do símbolo Domingo.
```

Além disso o enum **sem escopo** permite uma conversão **implícita** para **int**:

```
int var = Domingo; // e var receberá o valor 0 (zero) – por conversão implícita
```

2) Mas, a partir de C++11, podemos declarar um **enum com escopo próprio**, através da diretiva **"enum class"**:

```
enum class Semana { Domingo, Segunda, Terca, Quarta, Quinta, Sexta, Sabado } ;
int Domingo = 5; // OK! o "int Domingo" é um símbolo diferente de "Semana::Domingo"
```

3) Por último, como vimos acima, o enum sem escopo do C++98 permite uma conversão implícita para **int**.

Mas o enum com escopo do C++11 (enum class) **não** permite conversões implícitas:

```
enum class Semana { Domingo, Segunda, Terca, Quarta, Quinta, Sexta, Sabado } ;
// int var = Domingo; // ERRO: Aqui não foi declarado nenhum símbolo "Domingo";
// e sim "Semana::Domingo"
// int var = Semana::Domingo; // Outro ERRO: "Semana::Domingo" existe,
// mas o "enum class" não permite conversões implícitas
int var = int(Semana::Domingo); // OK! conversão explícita.
```

### 4.4.3.3 • Definindo o tipo inteiro subjacente de um enum (C++11)

A partir de C++11, podemos determinar o tipo inteiro **subjacente** que terá cada constante de um **enum**.

Isso é feito da seguinte forma:

```
enum : char { const_1, const_2 } ; // o tipo subjacente de const_1 e const_2 é char
enum Consts : short { const_1, const_2 } ; // tipo subjacente: short
enum class Consts : long { const_1, const_2 } ; // tipo subjacente: long
```

Esse tipo deve **sempre** ser um **inteiro**. Por exemplo:

```
// enum : double { const_1, const_2 } ; // ERRO! o tipo subjacente não é um inteiro
```

### 4.4.3.4 • Exemplos de escopo e especificação do seu tipo inteiro

```
#include <iostream>
```

```
int main()
```

```
{
```

```
    std::cout << "\n testEnum\n\n";
```

```
    enum : char { const_1, const_2};
```

```
    std::cout << "sizeof const_1 = " << sizeof(const_1) << '\n';
```

```
    // ➔ Imprime: sizeof const_1 = 1 // (que é o tamanho do 'char')
```

```
    // enum : double { c1, c2}; // ERRO: o enum só pode comportar um tipo inteiro
```

```
{
```

```
    // enum com tipo mas sem escopo
```

```
        // (com ou sem especificação do seu tipo inteiro subjacente):
```

```
    std::cout << "\n -- enum com tipo SEM escopo\n";
```

```
    enum Semana : short { Domingo, Segunda, Terca, Quarta, Quinta, Sexta, Sabado } ;
```

```

std::cout << "sizeof enum Semana = " << sizeof(Semana) << '\n'; // refere-se a
// "Semana: " (do enum Semana sem escopo);
// ➔ Imprime: sizeof enum Semana = 2 // (que é o tamanho do 'short')
int var = Domingo; // conversão implícita de "Semana" (sem escopo) para "int"
std::cout << "var (=Domingo) = " << var << '\n'; // imprime 0
// int Domingo = 5; // ERRO: redefinição de "Domingo", já que "Domingo"
// (do enum Semana) não tem escopo próprio
}
{
// a partir de C++11: enum com tipo e com escopo
// (com ou sem especificação do seu tipo inteiro subjacente):
std::cout << "\n – enum com tipo e COM escopo\n";
enum class Semana : long { Domingo, Segunda, Terça,
                          Quarta, Quinta, Sexta, Sabado } ;
std::cout << "sizeof enum class Semana = " << sizeof(Semana) << '\n';
// ➔ Imprime: sizeof enum class Semana = 4 // (que é o tamanho do 'long')
// int var = Domingo; // ERRO: neste escopo não existe nenhum símbolo "Domingo"
// abaixo, OK: resolve o escopo de Domingo (Semana::Domingo)
int var = int(Semana::Domingo); // conversão explícita de "Semana" para "int"
std::cout << "var (=Semana::Domingo) = " << var << '\n'; // imprime 0
int Domingo = 5; // "int Domingo": não tem nada a ver com "Semana::Domingo"
std::cout << "int Domingo = " << Domingo << '\n' ; // refere-se ao "int Domingo" : 5
}
std::cout << '\n';
return 0;
}

```

#### 4.4.4 • Constantes nomeadas em um único lugar

Finalmente, cabe observar que há ainda uma outra vantagem em associar valores constantes a nomes. Pois pode ocorrer que, no futuro, uma determinada constante venha a ser usada com um outro valor. E, se uma constante é nomeada em um único lugar no escopo em que será usada, não teremos muito trabalho para usar um novo valor.

Neste caso, bastará alterar a declaração da constante (que estará em um **único lugar**), ao invés de ter que alterar diversas linhas espalhadas pelo código, o que precisaria ser feito caso estivéssemos usando o valor constante bruto diretamente.

##### Exemplo clássico:

```

const double pi = 3.14159 ;
// ... // "pi" é usado a partir daqui ao invés de "3.14159".

```

Desse modo, no restante do código, usamos o nome "**pi**", ao invés do valor bruto.


**No futuro**, talvez seja necessário que essa constante seja usada de modo **mais preciso**. Nesse caso, bastará mudar a sua **única linha** de declaração:

```

const double pi = 3.14159265359 ;
// ... // E nada resta a alterar daqui para baixo. Isso é muito bom... 👍

```

---


 **Ressaltando:** constantes são **inicializadas obrigatoriamente em um único lugar** no escopo em que são declaradas. E **não** poderão ser **alteradas** através de atribuição (do contrário não seriam constantes...). **Então:**

---

**pi = 3.14 ;** //  Tentativa de **atribuir** valor a uma **constante**. **Erro** de compilação.

## 4.5 • Declarações auto (C++11)

No C++11 podemos declarar uma variável usando a palavra reservada **auto**. Isso pode evitar que o desenvolvedor se esqueça de inicializar uma variável, uma vez que qualquer **declaração auto** deve ser **inicializada para que o compilador possa deduzir o tipo** da variável ou constante.

 Se **não** inicializarmos uma declaração **auto**, o compilador emitirá uma mensagem de **erro**, pois **não conseguirá deduzir o tipo** da variável a partir do tipo do seu **valor de inicialização** (que é a sua **definição**). **Exemplos:**

```
#include <iostream>
int main()
{
    auto a = 1; // "a" é do tipo int que é deduzido do valor de inicialização 1 (int)
    auto b = "AGIT"; // "b" é do tipo string
                    // - também deduzido do valor de inicialização
    const auto c = 2; // o mesmo que foi dito para "a", mas "c" é "const"
    const auto d = "INFORMATICA"; // o mesmo que foi dito para "b" (mas "const")

    // auto e; // Erro: auto não inicializada (impossível deduzir o tipo de 'e')
    // const auto f; // Erro: auto (e const!) não inicializada...
    std::cout << a << ", " << b << ", " << c << ", " << d << '\n';
    return 0;
}

// Resultado: 1, AGIT, 2, INFORMATICA
```

## 4.6 • Conceitos: valores, tipos e objetos

Bjarne Stroustrup, em "**Programming: Principles and Practice Using C++**" (seção 3.8), estabelece algumas distinções úteis para melhor caracterizar o uso da memória:

- Um **tipo** define um intervalo de valores possíveis e um conjunto de operações (para um objeto).
- Um **objeto** é uma determinada área de memória que contem um valor de um determinado **tipo**.
- Um **valor** é um conjunto de *bits* na memória, o qual é interpretado de acordo com o seu **tipo**.
- Uma **variável** é um objeto que tem um **nome**.
- Uma **declaração** é uma instrução que dá um **nome a um objeto**.
- Uma **definição** é uma declaração que **aloca memória para um objeto**.

Mais acima, muitas vezes nos referimos ao uso de "memórias" de maneira genérica. Mas dissemos também que uma memória precisa ser **qualificada** (com um **tipo**, tornando-se um "objeto") para que possa ser usada.

E a questão central para sua qualificação é o seu **tipo**. Sem isso, não sabemos quais **valores** poderão ocupar legitimamente essa área de memória (e já vimos que números reais **não devem** ser armazenados como inteiros).

Em segundo lugar, decidimos se essa área de memória poderá ser alterada (**variável**) ou não (**constante**).

Quando **declaramos** uma variável ou uma constante, associamos um **nome** a uma determinada área de memória, a qual está sendo reservada de acordo com o **tipo** que rege

a declaração e que define o seu **tamanho** (a quantidade de memória a ser alocada) e a sua **forma de armazenamento**.

Os **valores** que serão armazenados em uma área de memória já declarada estarão assim qualificados com o **tipo** empregado nessa declaração. Por isso podemos pensar nesses valores como **objetos**: não são apenas um conjunto arbitrário de *bits*, mas um conjunto qualificado, isto é, com características próprias.

## 4.7 • Conversões entre tipos

Por enquanto, vamos apenas introduzir o assunto, chamando a atenção para alguns problemas potenciais que podem ocorrer quando trabalhamos com tipos diferentes que às vezes precisam estar presentes em uma mesma operação ou em operações encadeadas.

Para isso, vamos iniciar ampliando a nossa **tabela de tipos** na tabela abaixo (a tabela completa está no “guia de consulta rápida”, seção **1**, página **477**):


Mais alguns tipos, com seus tamanhos e intervalo de valores:			
Tipo	Tamanho mínimo em bytes	Intervalo de valores	Forma de armazenamento / natureza
<b>unsigned char</b>	1	0 a 255	inteiro
<b>char</b>	1	-128 a 127	inteiro
<b>unsigned short</b>	2	0 a 65535	inteiro
<b>short</b>	2	-32768 a 32767	inteiro
<b>unsigned long</b>	4	0 a +4294967295	inteiro
<b>long</b>	4	-2147483648 a +2147483647	inteiro
<b>unsigned int</b>	<i>depende da plataforma</i>		inteiro
<b>int</b>	<i>depende da plataforma</i>		inteiro
<b>unsigned long long</b>	64 bits sem sinal		inteiro
<b>long long</b>	64 bits		inteiro
<b>std::size_t</b>	<ul style="list-style-type: none"> <li>- <b>pode</b> ser equivalente ao <b>unsigned int</b> (em plataformas de <b>32 bits</b>);</li> <li>- ou <b>pode</b> ser equivalente ao <b>unsigned long long</b> (plataformas de <b>64 bits</b>)</li> </ul>		inteiro (usado para <b>dimensões</b> em geral e <b>índices</b> de vetores).
<b>enum</b>	De <b>1 byte</b> até, no máximo, o tamanho que tenha o <b>int</b> , dependendo dos <b>valores usados</b> . Em <b>C++11</b> <b>deve</b> ser <b>int</b> .		<b>inteiro</b> (enumera <b>constantes</b> )
<b>enum : &lt;tipo_inteiro&gt;</b>	Assume o indicado em <b>&lt;tipo_inteiro&gt;</b> , que deve ser um <b>inteiro (C++11)</b>		inteiro (constantes) - <b>C++11</b>
<b>bool</b>	1	<b>true / false</b> (1 ou zero)	<b>inteiro</b> (só <b>C++</b> )
<b>float</b>	4	3.4E-38 a 3.4E+38	<b>ponto flutuante</b>
<b>double</b>	8	1.7E-308 a 1.7E+308	<b>ponto flutuante</b>

Como podemos observar na tabela acima, temos:

- números **inteiros de diferentes tamanhos mínimos**: **char** (1 *byte*), **short** (2 *bytes*), **long** (4 *bytes*) e o **int** (cujo tamanho depende da plataforma: 2 *bytes* em máquinas de 16 *bits*, 4 em máquinas de 32 *bits*, etc), **long long** (64 *bits*)
- o tipo **bool** (1 *byte*), que, na memória irá armazenar os valores **1** e **0** - mas no código fonte devem ser expressos com as palavras reservadas **true** e **false**.
- **números reais de diferentes tamanhos**: o **float** (4 *bytes*) e o **double** (8 *bytes*).

Além dos diferentes tamanhos para cada forma de armazenamento, temos ainda um outro diferencial: observe que para os tipos inteiros (exceto o *bool* e o *enum*) temos uma repetição de cada um deles **precedida por um modificador**: **signed**.

Portanto, **tome nota**:

 Os tipos inteiros (exceto o *bool* e o *enum*), admitem duas alternativas, indicadas por **dois modificadores de tipo** que **antecedem** o tipo:



- **signed**: com sinal, admite números negativos e positivos; este é o **default**, logo pode ser omitido; ou seja: **basta** declarar "**int**" ao invés de "**signed int**".
- **unsigned**: sem sinal, apenas positivos.
- o **enum** **não** pode ser **precedido** por **signed** ou **unsigned**. Mas, a partir de **C++11**, podemos indicar seu tipo inteiro **subjacente**: **enum : unsigned int { /\*...\*/ }**;



Já os números de **ponto flutuante** são **sempre signed** – com sinal: **não podem** ser declarados como **unsigned**.

Cada tipo tem sua aplicação de acordo com a situação em que são usados. Mas é preciso cuidado quanto é preciso relacionar **tipos diferentes** em determinadas operações.

Exemplos:

<code>bool flag = true ; std::cout &lt;&lt; flag ;</code>	<b>imprime: 1</b> { OK: <b>true</b> é convertido para <b>1</b> , para que a impressão seja possível (conversão natural). }
<code>short i = 2000 ; std::cout &lt;&lt; i ;</code>	<b>imprime: 2000</b> { OK: imprime exatamente o esperado. }
<code>flag = i ; std::cout &lt;&lt; flag ;</code> (  )	<b>imprime: 1</b> { Ao converter <b>short</b> para <b>bool</b> [ <b>flag = i</b> ], o inteiro <b>2000</b> foi <b>truncado</b> para <b>true</b> , já que 2000 é <b>diferente de zero</b> , logo é <b>verdadeiro</b> . A conversão é natural, mas será que era isso que esperávamos? }
<code>i = flag ; std::cout &lt;&lt; i ;</code>	<b>imprime: 1</b> { OK: [ <b>i = flag</b> ], <b>true</b> é convertido para <b>1</b> . }
<code>char c = 65 ; std::cout &lt;&lt; c ;</code>	<b>imprime: A</b> { OK: 'A' é a representação gráfica do número 65 na tabela ASCII. E <b>cout</b> , ao receber um <b>char</b> , entende que deve imprimir sua representação gráfica (isto é, o caractere correspondente na tabela de <b>caracteres</b> em uso). }
<code>i = c ; std::cout &lt;&lt; i ;</code>	<b>imprime: 65</b> { Nada a ressaltar: o valor continua <b>o mesmo</b> . Apenas o <b>cout</b> , recebendo um <b>short</b> , imprime o próprio número ao invés de imprimir sua representação gráfica. }
<code>i = 323 ; c = i ; std::cout &lt;&lt; c ;</code> (  )	<b>imprime: C</b> { Ao converter <b>short</b> para <b>char</b> [ <b>c = i</b> ], o número <b>323</b> foi <b>truncado</b> para <b>67</b> , já que <b>323</b> não pode ser armazenado em um único byte (tamanho do <b>char</b> ). E <b>cout</b> , ao receber um <b>char</b> , imprimiu a representação gráfica de <b>67</b> : ' <b>C</b> '. A conversão é natural, mas era isso que esperávamos? }
<code>i = c ; std::cout &lt;&lt; i ;</code>	<b>imprime: 67</b> { Nada a ressaltar: o valor continua <b>o mesmo</b> . }

(👉) Nesses dois casos, conversão de **short** para **bool** [ **flag = i** ] e conversão de **short** para **char** [ **c = i** ], temos uma conversão natural entre tipos inteiros, mas com um problema: o **maior** (**short**, 2 bytes) está sendo convertido para o **menor** (**bool**, ou **char**, ambos com 1 byte, sendo que os valores admitidos pelo **bool** são 0 e 1).

Em conversões assim (maior para menor) os números poderão ser truncados quando (como ocorre nesses dois exemplos) os valores atuais dos objetos de tipo maior não estejam dentro dos intervalos de valores dos objetos de tipo menor.

Todos (ou quase todos) os compiladores admitem que os chamemos passando argumentos que elevam o seu "nível de **warning**" (ou advertência), de modo a obter uma análise mais rigorosa do código. E, em casos assim, alguns deles emitirão mensagens de advertência, indicando uma **possível perda de dados**. *Como, por exemplo:*

**warning :conversion from 'short' to 'char', possible loss of data**



**Mas, infelizmente, isso não é exigido pelo padrão da linguagem para os tipos primitivos e, desse modo, alguns compiladores ficarão em silêncio em casos desse tipo.**

Por isso o programador deve ser sempre explícito quando estiver realizando conversões de tipos maiores para menores, deixando clara a sua intenção. *Por exemplo:*

<b>flag = bool( i );</b>	Conversão <b>explícita</b> .
<b>c = char( i );</b>	Idem.

Isso não altera a **natureza** do problema, mas **deixa claro** que não houve uma distração do programador e sim uma ação **deliberada** (intencional). Quando alguém precisar ler o código isso fará diferença.

Uma outra situação, já descrita acima, seria esta:

```
int x ;
x = 5.3 ;           // 5.3 é do tipo double
```

Uma conversão de **double** para **int** é ainda **mais suspeita**, por envolver **diferentes formas de armazenamento**. Neste caso, felizmente, todos (ou quase todos) os compiladores emitirão um aviso semelhante a este:

**warning: converting to 'int' from 'double'**

Outra situação semelhante: **int x ; unsigned int y ;**  
**if ( x > y )**

// comparando **signed** com **unsigned**

Uma comparação entre inteiros com sinal e inteiros sem sinal pode levar a resultados inesperados, já que os seus intervalos de valores são diferentes. Aqui **podemos** ter um aviso: **warning: comparison between signed and unsigned integer**.

Além disso, nas seguintes operações: **double d = double ( 4 ) \* 1024 \* 1024 \* 1024 ;**

- Considere que os números constantes **4** e **1024** por *default* são do tipo **int**. Já, por exemplo, **4.0** (ou apenas **4.**) é, também por *default*, do tipo **double**.
- Quando fazemos: **double( 4 )**, estamos convertendo uma constante **int** para **double**, e com isso o resultado das operações de multiplicação também será do tipo **double**, pois ele é maior (ou **mais preciso**) que o **int**.
- Se não fizéssemos essa conversão, o resultado seria **int**. Em plataformas de 32 bits, onde esse tipo tem quatro bytes (ou seja, o mesmo tamanho e intervalo de valores do **long**), o resultado final seria **truncado (por overflow)** de 4.294.967.296 para -2.147.483.648, pois estaria estourando o valor máximo suportado.



- Em seguida, ocorreria a operação de atribuição para um **double** ("d"), que suportaria armazenar 4.294.967.296.
- Mas, **se o resultado da multiplicação estiver truncado**, será armazenado o **valor truncado**.

Por enquanto, o nosso objetivo é apenas advertir sobre problemas desse tipo. Contudo, esse assunto merece uma atenção maior, e será abordado em detalhes mais a frente. Em especial, conversões como [ **c = char( i )** ] deveriam ser implementadas de modo mais claro. Como veremos mais tarde, C++ oferece recursos mais explícitos para conversões.

## 4.8 • Variáveis classificadas como *static* e *extern*

A palavra reservada *static* indica que uma determinada memória **conserva** o seu valor. Temos 3 tipos de variáveis *static*. Inicialmente veremos **2 delas** (pois a terceira depende de um recurso que só veremos no capítulo 7, página **208** - "*Iniciando orientação a objetos*").

### 4.8.1 • Variáveis *static* de um bloco

No interior de qualquer bloco (inclusive uma função) podemos declarar uma variável como *static*.

**Exemplo:**

```
void func()
{
    int a = 5; // será sempre alocada a cada nova chamada desta função e inicializada.
    static int b = 1; // será inicializada uma única vez e depois conservará o último valor.
    // .....
    std::cout << "a (local) = " << a << ", b(static) = " << b;
    // .....
    a = 10;
    b = 15; // último valor atribuído à static "b"
    //.....
}
```

Toda a vez que "func" é chamada, a variável "a" (que **não é static**) é **reinicializada** com o valor 5. Embora depois receba o valor 10, "a" será **liberada** ao fim da função (retorno). Logo, se "func" for chamada novamente, "a" terá que ser novamente **alocada**, e, consequentemente, **inicializada com 5**.

Contudo "b" (que é *static*) **só é inicializada na primeira vez** em que "func" é chamada. A partir daí, **conservará o último valor** armazenado (**15**, no exemplo acima). Dessa forma a cada nova chamada de "func" **não ocorrerá** a inicialização ("b = 1").

Eis mais um caso em que a **diferença entre inicialização e atribuição** é crucial.

Dessa forma se chamarmos "func" 2 vezes, teremos a seguinte impressão:

```
func(); // primeira chamada
a (local) = 5, b(static) = 1
//....
func(); // segunda chamada
a (local) = 5, b(static) = 15 // "a" foi realocada e portanto reinicializada.
// mas "b" conservou o último valor recebido na chamada anterior
```

---

✎ Como "b" foi declarada como *static* dentro de uma função (escopo **local**), só poderá ser acessada no interior **dessa função**.  
Portanto, embora tenha **tempo de vida** permanente (ao contrário de "a", que tem tempo de vida local e temporário), sua **visibilidade é local** (do mesmo modo que "a").

---

O mesmo se aplica a **qualquer bloco** de código. *Exemplo:*

```
// .....
if ( condicao )
{
    static int x = 1;
    int y = 10;

    // .... possivelmente altera a static "x", que manterá o último valor armazenado.
} // "y" é desalocada aqui; mas "x" continua na memória com seu último valor

// Abaixo erro: "x" e "y" não são visíveis aqui - e não podem ser acessadas:
// fora do bloco de código em que foram declaradas:
std::cout << "x=" << x << ", y=" << y << '\n'; // ERRO!
// .....
```

## 4.8.2 • Variáveis *static* de um módulo e variáveis *extern*

Em um módulo (isto é, **um arquivo fonte**, formalmente denominado "*unidade de tradução*") também podemos declarar tanto variáveis da classe *static* como variáveis globais (classe *extern*).

Nos dois casos as declarações devem ser feitas **fora de qualquer bloco** (e, portanto, fora de qualquer função). *Exemplo:*

```
// arquivo "a.cpp"

static int st_var = 0; // uma variável static externa (declarada fora de qualquer bloco);
// só poderá ser acessada nas funções deste arquivo, abaixo desta declaração.

int g_var = 0; // uma variável global ou extern (declarada fora de qualquer bloco)
// poderá ser acessada por funções deste e de outros arquivos.
```

Além disso também podemos declarar **funções** como *static*. E isso significa que tais funções só poderão ser acessadas por outras funções presentes no arquivo em que as funções *static* são declaradas e implementadas (sempre abaixo de sua declaração).

*Continuando o exemplo:*

```
// arquivo "a.cpp"
#include <iostream>

static int st_var = 0; // variável static externa, específica deste arquivo.
int g_var = 0; // variável extern (global), pode ser acessada por toda a aplicação.

static void funcao_deste_arquivo() { /*.....*/ } // função static

void func_1() // função não-static, ou seja global (extern), pode ser chamada
// em qualquer função de qualquer arquivo da aplicação.
{
    std::cout << st_var << '\n'; // no mesmo arquivo pode-se acessar "st_var".
    std::cout << g_var << '\n'; // em qualquer arquivo pode-se acessar "g_var".
    funcao_deste_arquivo(); // em qualquer função do mesmo arquivo,
    // abaixo de sua declaração, pode-se chamar "funcao_deste_arquivo"
}
```

### 4.8.3 • Conformidade com C++



Atenção: variáveis e funções **estáticas de arquivo** (*static externas*) são um recurso de programação modular da linguagem C.

Em C++ podem ser usadas. Mas isso deve ser **evitado**, pois ambas são consideradas **oficialmente** como **obsoletas em C++**.

Isto porque C++ suporta **orientação a objetos**, que substitui programação modular. Veremos isso no capítulo 7, página 208 ("Iniciando orientação a objetos").



Já variáveis estáticas de **bloco não** têm qualquer restrição. Continuam **válidas** e são necessárias e úteis em certos casos.

## 4.9 • Como a memória é organizada

### 4.9.1 • Memória global ou estática

- É reservada, de início, uma área de memória para as variáveis globais (*extern*) e *static externas*.

Isto faz sentido, pois, como essas variáveis são **alocadas** no **início** da **aplicação** (antes de *main*) e só são **desalocadas** ao **final** da **aplicação** (após o retorno de *main*), essa área não precisará mais ser realocada. Essa é portanto a área **global** (também chamada de **estática**).

Seu **valor pode mudar**, mas sua **posição na memória** permanece **inalterada**.

- Assim sendo o **tempo de vida** dessas variáveis é global (a própria aplicação). Mas elas diferem quanto à **visibilidade**:
  - variáveis *static* têm visibilidade modular, isto é, só podem ser acessadas em funções presentes no **arquivo** em que foram declaradas (**escopo de arquivo**);
  - já variáveis *extern* têm visibilidade global, isto é, podem ser acessadas em qualquer função de qualquer arquivo da aplicação (**escopo da aplicação**).

### 4.9.2 • A pilha (*stack*)

- É reservada uma outra área para as variáveis locais (é a **pilha** ou *stack*). Isso se aplica tanto a **parâmetros** de função como a variáveis **alocadas dentro de um bloco** de código (**não-static**).

Dentro desta área haverá um movimento constante, pois teremos sempre variáveis sendo criadas e "morrendo" (isto é, sendo liberadas), conforme as funções são chamadas e em seguida retornam.

#### *Exemplo:*

```
void func (int y)
```

```
{
```

```
    int a=0;
```

```
    int b=0;
```

```
    //.....
```

```
} // "b", "a" e "y" são desalocadas aqui.
```

### 4.9.3 • Livre alocação

- A área restante é o **"heap"**. Essa é uma área de **livre** alocação: o programador pode assim solicitar que determinadas quantidades de memória desse trecho sejam alocadas em **tempo de execução**.

Mas, neste caso, o próprio programador deverá também **liberar** manualmente as memórias que tenha alocado. Veremos essa área de memória no capítulo 6, página 186 ("Ponteiros e referências").

## 4.10 • Vetores

Além dos tipos que já vimos, temos outros tipos. Dizemos que tipos suportados diretamente pela CPU são **tipos primitivos**. Assim, teremos uma variedade de tipos inteiros de diferentes tamanhos, e o mesmo para os números reais.

A linguagem também permite que o programador crie **novos tipos** a partir de tipos já existentes. Veremos como **criar** novos tipos mais a frente. Por enquanto, vamos **usar** alguns dos tipos suportados pela biblioteca padrão de C++ (**não** existem em C).

### 4.10.1 • Vetores: o que são e para que servem

Em muitas situações precisamos não apenas de um único **int** ou **double**, mas sim de uma série deles.

#### Exemplo:

Queremos armazenar os valores referentes aos totais das vendas de cada mês. Se alocaarmos esses valores individualmente, teremos uma sobrecarga de código ao acessá-los:

```
double vendas_mes_1 = 1000;
double vendas_mes_2 = 1200;
//...
double vendas_mes_12 = 1800;
```

No futuro, teremos que acessar **sempre** cada elemento individualmente, mesmo que o acesso seja repetitivo. Isso multiplicará a quantidade de instruções de acesso.

Se precisarmos imprimir esses 12 valores, teremos **12 instruções** de impressão individuais. Isso só irá **piorar** se a quantidade de valores aumentar.

#### Por exemplo:

- Se ao invés dos 12 meses, tivéssemos que imprimir os totais de vendas de cada semana de um ano, teríamos então mais de 50 variáveis individuais - e, para acessá-las, precisaríamos dessa quantidade de instruções, seja para impressão ou outras operações que poderiam ser executadas repetitivamente em um **laço**.

Um **vetor** permite que valores como esses formem uma **série**, de tal modo que podemos percorrer essa série em um laço, acessando cada um de seus elementos a cada nova iteração do laço com uma única instrução (uma única impressão, por exemplo) ou um único bloco de instruções.

Podemos resolver isso com dois tipos oferecidos pela biblioteca C++: **string** e **vector**.

### 4.10.2 • std::string e std::vector

- std::string** - armazena e permite o tratamento de um vetor de caracteres (uma cadeia de caracteres). Embora no fundo seja um vetor, esse tipo diferenciado permite que tenhamos funções especializadas em tratar cadeias de caracteres, que são muito comuns em programação (textos, principalmente).
  - Por exemplo, se fizermos:

```
std::string nome = "MARIA";
```

**Teremos na memória algo como:**

valor armazenado:	M	A	R	I	A	<TZ>
posição:	0	1	2	3	4	5

- Onde **<TC>** é um caractere especial de **terminação** da **string** (e, por convenção, esse terminador é o zero binário), de modo que:

```
std::cout << nome ;
```

**Irá imprimir "MARIA".** Ou seja: imprimirá todos os caracteres até encontrar **<TZ>** (o terminador zero), interrompendo aí a impressão.

- A **posição** é um índice **sequencial** (iniciado em **zero**) que permite acessar cada elemento individualmente, de modo que:

```
std::cout << nome[ 2 ] ;
```

**Irá imprimir o caractere 'R'.**

- **std::vector** - armazena e permite o tratamento de um vetor de **qualquer tipo** já existente, desde que esse tipo **admita a operação de atribuição**, pois cada elemento do vetor pode receber uma cópia de um determinado valor desse tipo.
- Por exemplo, para armazenar os 5 primeiros números primos:

```
std::vector< int > primos;
```

**Vetor de inteiros: tipo** indicado com **<int>**

```
std::size_t pr_max= 5;
primos.resize( pr_max );
```

**std::size\_t**: a dimensão não pode ser negativa.

**Dimensiona**: aloca memória para **5** inteiros.

Observar que [ **primos.resize( pr\_max );** ] indica que **std::vector** (ao contrário dos tipos primitivos) nos fornece **funções** especializadas em trabalhar com esse tipo. Veremos isso no capítulo **7**, página **208**.

```
primos [ 0 ] = 2 ;
primos [ 1 ] = 3 ;
primos [ 2 ] = 5 ;
primos [ 3 ] = 7 ;
primos [ 4 ] = 11 ;
```

**Acessa cada elemento**, usando o índice:

**primos [ <índice> ]** → **usa o "operador [ ]"**

O índice é baseado em zero. Se o vetor tem **5** elementos, podemos usar índices de **0** até **4**.

- Em consequência, teremos na memória algo como:

valor armazenado:	2	3	5	7	11
posição:	0	1	2	3	4

- A convenção usada para **strings**, onde temos um terminador, é útil apenas para textos (que agrupam caracteres), e por isso não se aplica a vetores em geral.

- Desse modo, em princípio, um vetor deve ser controlado pelo seu **tamanho**, e não por um terminador especial.
- Pois, **exceto em textos**, o zero binário pode ser um elemento qualquer do vetor (e não seu terminador).
- Então, se quisermos imprimir todos os elementos do vetor, uma das alternativas seria a seguinte:

```
size_t indice ; // size_t: o índice não pode ser negativo...
```

```
for ( indice = 0 ; indice < primos.size() ; indice = indice + 1 )
    std::cout << primos [ indice ] << " , " ;
```

E serão impressos **todos os elementos** do vetor [de **0** até **p\_max-1**] :

2 , 3 , 5 , 7 , 11 ,

- **std::vector** fornece-nos a função **size**, que retorna a dimensão do vetor.

- Tal como em **string**, a **posição** é um índice **sequencial** (iniciado em **zero**) que permite acessar cada elemento individualmente, coisa que, aliás, já fizemos no laço acima. Assim, também poderíamos fazer:

```
std::cout << primos [ 2 ] ;
```

**Irá imprimir o número 5.**

### Exemplo.

#### a. Criar um arquivo e escrever o código.

- Em um editor ou ambiente, dentro do diretório-base de exercícios ("**cursoCPP**"), crie um novo diretório para este exercício; por exemplo, "**03\_string\_vector**".
- Nesse diretório, grave o arquivo "**main.cpp**". Assim, teremos algo como:

**"cursoCPP/03\_string\_vector/main.cpp"**

Objetivo a atingir:

O código abaixo inicialmente usa um objeto **std::string**, e alguns exemplos muito simples de manipulação de texto.

Em seguida, cria um vetor (um objeto **std::vector**) para armazenar o total de vendas de cada trimestre de um ano. Esse tipo de valor exige centavos; logo será necessário o ponto flutuante e por isso usaremos o tipo **double**.

Assim teremos: **std::vector < double > vendas\_trimestre**. Esse vetor terá seus elementos preenchidos por entradas de dados a partir do teclado, em um laço.

Depois esses valores serão impressos e acumulados para gerar um valor anual - também em um laço.

```

C:\> Prompt de comando

=== a. testando string
Maria
Maria da Silva

=== b. testando vector
informe total de vendas de cada trimestre:
informe o trimestre : 1
100 ←-----informei...
informe o trimestre : 2
200 ←-----informei...
informe o trimestre : 3
300 ←-----informei...
informe o trimestre : 4
400 ←-----informei...

imprimindo os valores informados:
100
200
300
400
total vendas no ano : 1000
  
```

### Exemplo de implementação:

#### Código fonte

para "**main.cpp**":

```

#include <iostream>
#include <string>
#include <vector>

int main()
{
    // === a. string :

    std::cout << "=== a. testando string\n";

    std::string nome ;
    nome = "Maria" ;
  
```

```

std::cout << nome << "\n" ; // imprime [Maria]
nome = nome + " da Silva" ;
std::cout << nome << "\n" ; // imprime [Maria da Silva]

// === b. vetor :

std::cout << "\n=== b. testando vetor\n";

// um vetor para armazenar o total de vendas em um trimestre:
const std::size_t trimestres_ano = 4 ; // total de trimestres no ano...

// declara o vetor, que conterá uma série de objetos do tipo double:
std::vector < double > vendas_trimestres ; // tipo em uso: double

// define a quantidade de elementos do vetor
// (usando uma variável ou uma constante):
vendas_trimestres.resize( trimestres_ano ); // vetor para 4 elementos

// pede ao usuário que informe o total de vendas de cada trimestre:
std::cout << "informe total de vendas de cada trimestre:\n";
std::size_t index ;
for ( index = 0 ; index < trimestres_ano ; index = index + 1 )
{
    std::cout << "informe o trimestre : " << index + 1 << "\n";
    std::cin >> vendas_trimestres [ index ] ; // entrada para cada elemento
}

// imprime cada trimestre e totaliza os valores para o ano inteiro:
std::cout << "\nimprimindo os valores informados:\n";
double total_ano = 0;
for ( index = 0 ; index < trimestres_ano ; index = index + 1 )
{
    std::cout << vendas_trimestres [ index ] << "\n" ;
    total_ano = total_ano + vendas_trimestres [ index ] ;
}

// imprime o total do ano:
std::cout << "\ntotal vendas no ano : " << total_ano << "\n" ;
return 0;
}

```

**b. Compilar** o exercício: use ambientes de desenvolvimento ou a linha de comando.

**c. Executar** o programa. Informe os valores que serão solicitados. Reveja o exemplo de uso exibido acima, para comparar com o que você obteve.

Depois **modifique o código** com base nos exemplos de **string** e **vector**. Por exemplo, para imprimir **um único caractere da string**. Ou para testar o acesso a elementos **não alocados**.

### 4.10.3 • Vetores na linguagem C

Acima vimos um uso simples de **strings** e **vetores**, o qual é possível devido à biblioteca de C++.

Em algumas situações poderíamos usar, em C ou em C++, vetores "em baixo nível" (praticamente no nível da máquina). Aliás, essa é a única alternativa que temos no caso da linguagem C.

**Em C++ só devemos usá-los se:**

- Tiverem **quantidade fixa de elementos** (pois os vetores de C com quantidade variável de elementos apresentam alguns riscos adicionais, como veremos mais a frente). Tendo uma quantidade fixa, essa quantidade **não** deve ser grande.

- Além disso, só devem ser usados em situações onde, garantidamente, não existam riscos de acesso a elementos não alocados (fora da dimensão do vetor).

Considere os seguintes exemplos:

#### a. strings.

Em C++ fazemos:

```
std::string nome;
nome = "Maria";
nome = nome + " da Silva";
```

Usando o "estilo C", faríamos:

```
char nome[ 15 ]; // um vetor de 15 caracteres
strcpy ( nome , "Maria" );
strcat ( nome , " da Silva" );
```

- Dimensionamos um vetor de **15** caracteres (**char**).
- Em seguida, **copiamos** (função **strcpy**) a **string** "Maria" (constante) para esse vetor: [ **strcpy( nome, "Maria" );** ].
- Isso significa que todos os caracteres da constante são copiados, um a um, para os seis primeiros elementos do vetor (os cinco de "Maria", mais o caractere de terminação zero (<TZ>) que, por convenção, está embutido no final dessa constante):

valor:	M	a	r	i	a	<TZ>	(sem uso no momento)								
posição:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- Finalmente, usamos uma outra função (**strcat**) para concatenar uma segunda **string** ao vetor: [ **strcat( nome, " da Silva" );** ].
- Isso significa que todos os caracteres da **string** " da Silva" também serão copiados, um a um, para o vetor, a partir da posição onde está o terminador zero (após o último 'a' de "Maria").

valor:	M	a	r	i	a		d	a		S	i	l	v	a	<TZ>
posição:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Aparentemente nenhum problema. Todos os elementos do vetor estão dentro da dimensão máxima definida inicialmente: **15**.

Mas, se ao invés de:

```
strcat ( nome , " da Silva" );
```

tivéssemos:

```
strcat ( nome , " Aparecida" );
```

Estaríamos ultrapassando os 15 caracteres alocados para o vetor "nome" (e o mesmo poderia ocorrer com a função **strcpy**):

valor:	M	a	r	i	a		A	p	a	r	e	c	i	d	a	<TZ>
posição:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15


Isso **não** aconteceria com **std::string**, pois se fizermos:

```
std::string nome = "Maria";
```

**std::string** cuidará dos detalhes para que a memória seja redimensionada conforme as ne-



nome = nome + " Aparecida";	cessidades determinadas pelo uso.
-----------------------------	-----------------------------------

 Isso significa que o uso do **estilo "C"** para vetores (inclusive vetores de caracteres) exige **cuidados especiais**.

### b. vetores de qualquer tipo.

O mesmo problema observado com **strings** pode ocorrer com vetores em geral. Em um contexto controlado, podemos garantir um acesso seguro. Mas nem sempre isso é possível.

**No exemplo abaixo, não temos qualquer problema:**

```
const unsigned int primos_max = 5 ; // a constante nomeada servirá
// para permitir que a dimensão do vetor seja usada de modo coerente
// em todas as partes do código. Mas, isso dependerá do programador.
```

```
// Um vetor no estilo "C" pode ser inicializado com uma
// lista de inicialização entre chaves:
```

```
int primos [ primos_max ] = { 2 , 3 , 5 , 7 , 11 } ;
```

valor armazenado:	2	3	5	7	11
posição:	0	1	2	3	4

```
size_t i ;
```

```
for ( i = 0 ; i < primos_max ; i = i + 1 )
```

```
std::cout << primos [ i ] << " , " ;
```

Em uma situação assim, tudo está (ou parece estar) sob controle. Temos uma constante nomeada para determinar a dimensão do vetor e, no código acima, essa constante é sempre considerada para impedir que ocorra um acesso a um elemento não alocado.

Mas, isso não impede que, em outras partes do código, a "boa prática" seja esquecida e ocorra um acesso arbitrário. Tudo dependerá do cuidado do programador.

Na lista de inicialização, caso usássemos um sexto número, o compilador acusaria erro. Mas a partir daí estamos por nossa conta e risco. Temos que garantir, por todo o código, que não existirão tentativas de acesso a elementos não alocados.

**Por exemplo (e aqui temos problemas):**

<b>primos</b> [ 5 ] = 13 ;	acessa memória não alocada.
----------------------------	-----------------------------

Pois tudo o que temos são cinco elementos e não seis, como podemos ver nesta tabela:

valor armazenado:	2	3	5	7	11	13
posição:	0	1	2	3	4	5

O vetor foi dimensionado para **5** elementos. Logo, podemos acessá-lo com os índices de **0** até **4**. Ao usarmos o índice **5**, estamos tentando acessar um **sexto** elemento. Mas não temos memória reservada para acolher esse intruso.

Em consequência, é possível que o número **13** esteja sendo escrito em uma área de memória alocada para **outra finalidade**.

Os resultados desse tipo de erro são **imprevisíveis**, pois dependem de cada execução.

- Em determinada execução, escrevemos em uma área de memória que não está em uso (e, **na aparência**, não temos problemas).
- Em outra execução, escrevemos em uma área de memória **reservada para outro fim**, acarretando resultados inesperados.
- Há ainda uma terceira situação: sistemas multi-tarefa (como *Windows* ou *Unix/Linux/MacOs*) atribuem áreas de memória para cada aplicação que esteja em execução.
  - E, se determinada aplicação acessa uma área de memória que **não lhe foi cedida pelo SO**, a execução é interrompida e o sistema acusa uma tentativa ilegal de acesso à memória ("**segmentation fault**").

Bem, isso é ruim mas é **menos grave** do que escrever em uma memória cedida pelo SO mas que está **em uso para outra coisa...**

Pelo menos, ficamos sabendo rapidamente que algo está errado. Ao passo que se escrevemos em memórias destinadas a **outros fins**, a aplicação poderá continuar rodando e gerando danos. Apenas mais tarde é que perceberemos que temos problemas. E nem sempre será fácil detectar que tudo começou com aquele mísero "**13**".

Já se estivéssemos usando **std::vector**, poderíamos evitar isso.

<b>Se fizéssemos:</b>	<pre>std::vector&lt; int &gt; primos; primos.resize( 5 ); primos [ 5 ] = 13 ; // 🚫</pre>	<b>também teríamos o mesmo problema.</b>
-----------------------	--	--

✎ **Contudo, std::vector** oferece **outros** meios de acesso aos elementos do vetor além do **operador [ ]**: a função **at** e também a pré-verificação da dimensão com a função **size**. Com elas podemos ter um uso seguro de vetores.

✎ **Detalhes importantes** que precisaremos examinar:

- vetores com **dimensões variáveis**: são alocados na área de memória de **livre alocação** (o "heap"); **std::vector** e **std::string** usam essa área.

- vetores de **dimensões fixas**, mas muito **grandes**: se alocados **na pilha** podem levar a um estouro de pilha ("**stack overflow**"); portanto devem ser alocados na memória de **livre alocação**; mas **não** devemos fazer isso manualmente e sim usar **std::vector** ou **std::string** (conforme o caso).



Só use vetores no estilo "**C**" com **dimensões fixas (constantes)** e que usem **pouca memória**, em **contextos** absolutamente **seguros**.

✎ **Na dúvida, use std::string e std::vector**. Além do que já foi dito, eles oferecem um conjunto de recursos que não encontraremos, nativamente, em vetores no estilo **C**.

Veremos mais detalhes sobre os seus usos no capítulo **9**, página **273**.


#### 4.10.4 • Usos seguros de índices em vetores

O acesso a vetores através do operador **[]** (**índice**) é completamente **seguro** se o índice for **pré-verificado** em relação à **dimensão** do vetor.

✎ **Lembrando:** para **dimensões** e **índices** de vetores, usamos o tipo **std::size\_t** (sempre um inteiro **unsigned**: dimensões e índices **não** podem ser **negativos**). O **size\_t** fornece tanto **clareza** como também **portabilidade**. **Por exemplo:**

✎ - em **32 bits** o "**std::size\_t**" **pode** ser um sinônimo para "**unsigned int**";

---

 - mas, em **64 bits**, o "std::size\_t" **pode** ser um sinônimo para "unsigned long long" (ou equivalente, dependendo da plataforma).

---

### Exemplos:

#### - Usando std::vector:

```
std::vector<int> v1;
v1.resize(5);
size_t v1_size = v1.size();
for (size_t index=0; index < v1_size; ++index)
{
    v1[index]= 1001+int(index);
    std::cout << v1[index] << ", ";
}
std::cout << "\n"; // Resultado: 1001, 1002, 1003, 1004, 1005,
```

#### - Usando o "estilo C":

```
const std::size_t v2_size = 5;
int v2 [ v2_size ];
for (size_t index=0; index < v2_size; ++index)
{
    v2[index]= 2001 + int(index);
    std::cout << v2[index] << ", ";
}
std::cout << "\n"; // Resultado: 2001, 2002, 2003, 2004, 2005,
```

#### - Os dois tipos usando um if:

```
int x = 10;
//.....
if (x < v1.size() ) // "v1" é um std::vector< int >
    v1[ x ] = 30;
else
    std::cout << "tentativa de acessar elemento não alocado em v1\n";
if (x < v2_size ) // "v2" é um vetor para int no "estilo C"
    v2 [ x ] = 30;
else
    std::cout << "tentativa de acessar elemento não alocado em v2\n";
```

Em conclusão: em todos os exemplos acima o uso do **índice** ("operador [ ]") é totalmente **seguro**. Nos exemplos em que o **for** é usado ele **inicia em zero** e progride até "**size – 1**" (**index < size**). Logo, o acesso aos elementos dos vetores é feito por um índice que está **pré-verificado** pela própria **condição** do laço **for**. E o mesmo se aplica aos exemplos em que o **if** é usado para a pré-verificação.

## 4.10.5 • Caracteres: aspas simples e duplas

Por convenção, uma cadeia de caracteres constantes é representada por **aspas duplas**, incluindo sempre um caractere de terminação(zero binário, ou "terminador zero") ao seu final. Já um único caractere constante é representado por **aspas simples** e não inclui qualquer caractere de terminação pois é um valor inteiro único e **não** um vetor.

**Assim, se fizemos** (usando **aspas duplas**):

```
std::string nome = "Maria" ;
```

OU

```
char nome[ 6 ] = "Maria" ;
```

Teremos uma cadeia de caracteres (um vetor de elementos do tipo **char**) à qual é adicionada o "terminador zero":

valor:	M	a	r	i	a	0 (TZ)
posição:	0	1	2	3	4	5

**Já se fizemos**  
(usando  
**aspas simples**):

```
char c = 'M' ;
```

Temos uma única posição de memória para **um único valor inteiro do tipo char**.  
E nenhum caractere extra é alocado.

## 4.11 • Revisão do Capítulo 4

### 4.11.1 • Exercício

**Tente** resolver o exercício abaixo. **Em seguida, compare sua solução** com a que está no **Anexo B-4-1, página 436**. **Caso não entenda**, fale com o instrutor.

#### Enunciado:

- Crie um **vetor de inteiros** que deverá armazenar todos os números entre um número **inicial** e um **final**, existindo entre eles uma distância denominada "**razão**".
- O "inicial", o "final" e a "razão" devem ser **informados pelo usuário**.
- Após o armazenamento dos números no vetor, o programa deve **imprimir todos os elementos do vetor**.
- Esses números **também devem ser inseridos em uma string**, formando uma cadeia de caracteres. Por isso, o intervalo entre 'inicial' e 'final' deve ser **compatível** com o tipo **char** (por exemplo, entre 'A' e 'Z').
- Ao final, **imprimir**: o **total de números** no intervalo, a **soma dos elementos** do vetor e a **string** formada pelos números.
- O programa deve também permitir que o usuário **repita a operação**, informando novos valores.

**Resultado que deve ser impresso:**

```

C:\ Prompt de comando

progressao com caracteres

Informe Inicial, Final e Razao.
Limites: A <= Inicial <= Final <= Z.
Razao: maior que zero
e menor ou igual a <Final-Inicial+1>.

A
Z
12
65
77
89

Total de numeros no intervalo: 3
Resultado da soma: 231
String: AMY

deseja realizar novo calculo?
<0 para encerrar, 1 para continuar>
0
fim de processamento
  
```

#### Passos para atingir o objetivo:

- a. Usando um editor ou um ambiente, dentro do diretório-base de exercícios ("**cursoCPP**"), crie um novo diretório para este exercício; por exemplo, "**04\_tipos**". Em seguida, crie e salve nesse diretório um novo arquivo (por exemplo, "**main.cpp**"), de modo que teremos algo como:

/cursoCPP/04\_tipos/main.cpp

- b. Para começar, você pode copiar a função "**main**" do exercício do capítulo anterior: "**02\_lacos\_while\_for**" (e que está resolvido na página [431](#)). A estrutura será semelhante, com solicitações ao usuário dentro de um laço **while**, e também teremos "**inicial**", "**final**" e "**razao**". Mas serão necessárias algumas alterações.

- Antes de **main**, faça todos os **includes** necessários:
  - **iostream, limits, vector e string**.
- Em **main**, use **std::vector<int>** para criar o vetor de inteiros e **std::string** para criar uma string. Por exemplo:
 

```
std::vector<int> vec ;
std::string str ;
```
- Se for usar um *flag* para o laço de repetição de toda a operação (como, por exemplo, o "**while (continuar)**" do exemplo citado) defina essa variável como **bool** e não como **int**.
 

```
bool continuar = true;
while ( continuar ) { ... }
```

Mas **atenção**. Se fizermos:

```
std::cout << "Quer continuar?\n(0 p/encerrar, 1 p/continuar)\n";
cin >> continuar ;
```

Como a variável "**continuar**" agora é **bool**, **std::cin** só aceitará zero ou um.

Ao contrário do **int**, onde aceitávamos **zero** como falso ou **qualquer outro número** como verdadeiro, agora, se o usuário digitar um número diferente de 0(zero) ou 1(um), **std::cin** irá considerar a entrada inválida. Ficaremos sabendo disso assim:

```
if ( std::cin.fail() )
```

```
// Uma entrada inválida: qualquer coisa diferente de 0 e 1...
```

- Como vamos incluir os números também em uma **string**, o "inicial" e o "final" devem determinar um intervalo de valores compatível com o tipo **char**.
- Para isso, podemos definir os limites mínimo e máximo para 'inicial' e 'final', como, por exemplo:

```
const char inicial_min = 'A' ;
const char final_max   = 'Z' ;
```

Se garantirmos que os valores informados pelo usuário obedecerão a limites adequados, eles não serão negativos e, além disso, estarão dentro do intervalo de valores suportados pelo tipo **char**, e assim serão inseridos adequadamente na **string**. Para isso será preciso **analisar** as entradas de dados:

```
char inicial, final;
int razao;
std::cin >> inicial >> final >> razao ;
if ( inicial < inicial_min )
    // mensagem de erro - e "tente de novo".
if ( final > final_max )
    // mensagem de erro - e "tente de novo".
if ( inicial > final )
    // mensagem de erro - e "tente de novo".
```

- A distância entre cada número ("**razão**") também deve ser analisada:

- Deve ser **maior que zero**.
 

```
if ( razao <= 0 )
    // mensagem de erro - e "tente de novo"
```
- **Não** pode ser **maior que** [ **final - inicial + 1** ].
 

```
if ( razao > (final - inicial + 1) )
    // mensagem de erro - e "tente de novo"
```

- **Dimensione o vetor** e a **string** para a quantidade de números existentes no intervalo informado, sabendo-se que essa quantidade pode ser determinada assim:

```
// Quantidade total de números no intervalo:
unsigned int quant_num = (final-inicial+razao) / razao ;
```



**Obs.: a conversão** do resultado das operações aritméticas acima, o qual terá o tipo "**int**" (o tipo de 'razao') **para** a variável "**quant\_num**", que é "**unsigned int**", **seria insegura** caso pudesse ter um **valor negativo**. Além disso o resultado **não** pode ser **zero**, pois servirá para **dimensionar o vetor e a string**.

Mas, neste caso, nada disso acontecerá (e teremos uma conversão **segura**) se forem feitas todas as consistências indicadas acima.

Assim, se fizermos todas as análises relacionadas acima, teremos:

- A garantia de que 'inicial' e 'final' estarão dentro de um intervalo positivo e que 'inicial' não será maior que 'final'.
- A garantia de que 'razao' não pode ser maior que ('inicial-'final'+1) e desse modo o resultado da operação não será zero.
- Finalmente, garantimos também que 'razão' não pode ser zero (impedindo a divisão por zero).
- Assim sendo, você agora pode usar "**quant\_num**", de modo coerente e seguro, para **dimensionar o vetor e a string**. Por exemplo:

```
vec.resize( quant_num ); // Dimensiona o vetor (garantimos que
                          // a dimensão é maior que zero).
str.resize( quant_num ); // Dimensiona a string (idem).
```

- Em um laço **for**, **armazene no vetor** todos os números entre inicial e final (inclusive estes), e insira os mesmos na **string**. Por exemplo:

```
unsigned int index ;
for ( index = 0 ; inicial <= final ; inicial = inicial + razao ☹ )

// Mas isso pode levar a uma warning do compilador. É melhor fazer:
char r = char(razao) ; ☹

for ( index = 0 ; inicial <= final ; inicial = inicial + r ☹ )
```



Esta conversão visa evitar que o último segmento do laço **for** [ inicial = inicial + razao ] receba uma eventual **warning** do compilador, pois, 'razao' é **int**, e a atribuição será feita em 'inicial' que é **char**.

- Observe que, como "razao" é **int**, **poderia** conter um valor que, somado a 'inicial', não coubesse em um **char** - e a atribuição está sendo feita na própria 'inicial' que é **char**. Logo, teríamos: [ <char> (inicial) = <int> ].
- Acima, ao analisar e vetar valores incompatíveis, garantimos que **isso não poderá acontecer**, mas o compilador não é obrigado a analisar toda a lógica do programa. Ele apenas verifica uma **possível** truncagem de um determinado valor.
- Por isso, ao invés de [ inicial = inicial + razao ], fazemos [ inicial = inicial + r ], com **todos os tipos compatíveis**:

```
for ( index = 0 ; inicial <= final ; inicial = inicial + r ☹ )
{
    vec[ index ] = inicial ; // Conversão de char para int: segura.
    str [ index ] = inicial ; // Alimenta a string (OK: 'inicial' é char).
    index = index + 1; // Evoluir o índice.
}
```

- Em um novo laço **for**, **some e imprima os elementos do vetor**. Por exemplo:

```
for ( index = 0 ; index < vec.size() ; index = index + 1 )
{
    // somar vec [ index ] a um acumulador
    // imprimir vec[ index ]
}
```

- Ao final, antes de perguntar ao usuário se ele deseja continuar, **imprima**:
  - A quantidade de números no intervalo: **quant\_num**.
  - O resultado da **soma** acumulada no último laço **for**.
  - A **string** alimentada no primeiro laço **for**.

### c. Compilar

### d. Executar.



Compare o que você fez com a solução da apostila: página **436**.

### 4.11.2 • Questões para revisão

Responda às questões abaixo, assinalando **todas** as respostas corretas (**uma ou mais**). Em seguida, compare suas respostas com as respostas localizadas no **Anexo B-4-2, página 440**. Caso não entenda, encaminhe as dúvidas ao instrutor.

**Cap.4 - 1.** Assinale as respostas corretas, considerando estas declarações:

```
long i ;
float f ;
```

- a. ☐ 'i'(long) tem o mesmo **tamanho** de 'f'(float): 4 bytes. Logo podemos dizer que não há qualquer diferença quanto ao modo como essas variáveis serão representadas na memória.
- b. ☐ 'i'(long) tem uma **forma de armazenamento** diferente de 'f'(float), já que esta é representada na memória através de ponto flutuante. Logo, apesar de terem o mesmo tamanho, são representadas de modo **diferente**. Mas isso não tem **nenhuma** implicação prática.
- c. ☐ A forma de armazenamento de 'f'(float), através de ponto flutuante, torna o seu acesso mais lento do que o acesso a números inteiros.

**Cap.4 - 2.** Assinale as respostas corretas, considerando o código abaixo:

```
const int constante ;
//...
constante = 5 ;
```

- a. ☐ As duas linhas do código acima estão **corretas**.
- b. ☐ A primeira linha do código acima está **incorreta**, pois uma declaração **const** exige **inicialização**. A segunda linha está **correta**.
- c. ☐ A primeira linha do código acima está **incorreta**, pois uma declaração **const** exige **inicialização**. A segunda linha também está **incorreta**, pois constantes podem apenas ser inicializadas mas não podem sofrer **atribuições**.

**Cap.4 - 3.** Assinale as respostas corretas, considerando o código abaixo:

```
int variavel ;
//...
variavel = 5 ;
```

- a. ☐ As duas linhas do código acima estão **incorretas**.
- b. ☐ A primeira linha do código acima está **incorreta**, pois uma declaração de variável exige **inicialização** obrigatoriamente. A segunda linha está **correta**.
- c. ☐ As duas linhas do código acima estão **corretas**.

**Cap.4 - 4.** Assinale as respostas corretas, considerando estas declarações:

```
double d = 10.9 ;
int i = d ;
```

- a. ☐ O código acima não apresenta **nenhum tipo de problema**.



- b. ☐ A variável 'i'(int) não tem suporte a ponto flutuante. Logo o valor de 'd' (10.9) será truncado para 10 ao ser copiado para 'i'. Isso pode ser um problema, dependendo da situação em que ocorra.

**Cap.4 - 5.** Assinale as respostas corretas, considerando estas declarações:

```
double d = double (4) * 1024 * 1024 * 1024 ;
long i = d ;

// OBS.: o resultado de [ double (4) * 1024 * 1024 * 1024 ]
// é 4.294.967.296.
```

- a. ☐ O código acima não apresenta **nenhum tipo de problema**.
- b. ☐ O valor de 'd' (double) será truncado (por *overflow*) ao ser copiado para 'i' (long)

**Cap.4 - 6.** Assinale as respostas corretas, considerando o código abaixo:

```
std::string nome ;
nome = "Maria" ;
nome = nome + " Aparecida";
```

- a. ☐ As três linhas do código acima estão **corretas** e não implicam em **nenhum tipo de problema**.
- b. ☐ As três linhas do código acima estão **incorretas**..
- c. ☐ A terceira linha do código acima pode implicar em um problema, escrevendo em **memória ainda não alocada**.

**Cap.4 - 7.** Assinale as respostas corretas, considerando o código abaixo:

```
std::vector <int > vetor ;
vetor [ 0 ] = 10 ;
```

- a. ☐ As duas linhas do código acima estão **corretas** e não implicam em **nenhum tipo de problema**.
- b. ☐ As duas linhas do código acima estão **incorretas**..
- c. ☐ A primeira linha do código acima está **correta**. A segunda linha apresenta um **problema potencial**, já que "**vetor**" não foi dimensionada e há um acesso ao seu primeiro elemento ([0]). Antes disso, alguma coisa deveria ter sido feita, como, por exemplo: [ **vetor.resize( <dimensão> )** ] ; ]. Desse modo a execução da segunda linha implica em resultados **imprevisíveis** (ou indetermináveis).

**Cap.4 - 8.** Assinale as respostas corretas, considerando o código abaixo:

```
enum Meses { Janeiro=1, Fevereiro, Marco, Abril, Maio, Junho, Julho,
             Agosto, Setembro, Outubro, Novembro, Dezembro } ;

std::vector < std::string > vec ;
vec.resize( Dezembro ) ;
vec[ Janeiro ] = "Janeiro" ;
// faz o mesmo para Fevereiro, Marco... até vec[Dezembro]="Dezembro";

unsigned int index;
for ( index = 0 ; index < vec.size() ; index = index + 1 )
    std::cout << vec[ index] << "\n" ;
```

- 
- a. ☐ Todas as linhas do código acima estão **corretas** e não implicam em **nenhum tipo de problema**.
- 
- b. ☐ Todas as linhas do código acima estão **incorretas**.
- 
- c. ☐ **Não** é possível fazer: `[ std::vector < std::string > vec ; ]`.  
Só são possíveis os usos de `std::vector` para tipos básicos como:  
`[ std::vector < int > vi ; ]` ou `[ std::vector < double > vd ; ]`.
- 
- d. ☐ **Não** é possível fazer: `[ vec.resize( Dezembro ) ; ]`.
- 
- e. ☐ **Não** é possível fazer: `[ vec [ Janeiro ] = "Janeiro" ; ]`.
- 
- f. ☐ A expressão `[ vec.resize( Dezembro ) ; ]` está **correta** pois "**Meses**" pode ser convertido para **int**. Nesse caso, seria o mesmo que:  
`[ vec.resize( 12 ) ; ]`.
- 
- g. ☐ A expressão `[ vec [ Janeiro ] = "Janeiro" ; ]` está **correta** pois "**Meses**" pode ser convertido para **int**. Nesse caso, seria o mesmo que:  
`[ vec [ 1 ] = "Janeiro" ; ]`.
- 
- h. ☐ Caso não ocorra um erro de execução, o laço **for** do código acima irá imprimir:  
**Janeiro Fevereiro Marco, Abril ... Novembro Dezembro.**
- 
- i. ☐ Caso não ocorra um erro de execução, o laço **for** do código acima irá imprimir:  
**<uma string vazia>**  
**Janeiro Fevereiro, Marco, Abril ... Novembro Dezembro.**
- 
- j. ☐ Caso não ocorra um erro de execução, o laço **for** do código acima irá imprimir:  
**<uma string vazia>**  
**Janeiro Fevereiro Marco, Abril ... Outubro Novembro.**
- 
- k. ☐ É **possível** que ocorra um **erro de execução** na linha:  
`[ vec [ Dezembro ] = "Dezembro" ; ]`, ou em algum momento qualquer após sua execução.  
Isso porque o vetor foi **dimensionado** para **12** elementos e deve ser acessado com os índices de **0** a **11**.  
Mas, nessa linha, está sendo acessado com o **índice 12** - pois esse será o resultado da conversão da constante **Dezembro**, do tipo **Meses**, para **int**. Logo, está tentando acessar um **décimo-terceiro** elemento, para o qual não há memória alocada.  
Caso isso atinja uma área de memória não reservada para a aplicação em um sistema multi-tarefa, o SO irá interromper a aplicação.  
Do contrário, teremos resultados indeterminados em operações seguintes, pois é possível que essa operação esteja escrevendo em memória alocada, mas para outra finalidade (e este é o pior caso).
- 



**Compare o que você fez com a solução da apostila: página 440.**

## • Capítulo 5

### ▪ Fluxo de processamento

Neste capítulo iremos rever e aprofundar alguns dos tópicos vistos nos capítulos anteriores, acrescentando também novos elementos. Veremos:

- Detalhes sobre funções: parâmetros e retornos.
- O escopo de memórias em função do local de sua declaração.
- Operações e operadores: precedência de operadores.
- Mais informações sobre tomadas de decisão.
- E novos aspectos sobre laços.

5.1 • Tópicos de destaque neste capítulo.....	133
5.2 • Linhas de instrução; declarações e operações.....	135
5.3 • Operadores.....	136
5.3.1 • Acrescentando mais alguns operadores.....	136
5.3.2 • Operador condicional ternário.....	137
5.3.3 • Resto de divisão.....	138
5.3.4 • Operadores compostos.....	139
5.3.5 • Operações lógicas.....	141
5.3.6 • Em conclusão.....	143
5.4 • Precedência de operadores.....	143
5.5 • Finalização de uma linha de instrução.....	144
5.6 • Funções.....	144
5.6.1 • Parâmetros e valor de retorno de uma função.....	145
5.6.2 • Funções sem parâmetros e/ou valor de retorno.....	146
5.6.3 • A função main.....	147
5.6.3.1 • Retorno de main.....	148
5.6.3.2 • Parâmetros de main.....	149
5.6.4 • Protótipos de funções.....	150
5.6.5 • Arquivos <i>header</i> .....	151
5.6.6 • Escopo de uma função e escopo global.....	153
5.6.7 • Chamadas de função e seu custo.....	154
5.6.8 • Funções inline.....	154
5.6.9 • Funções recursivas.....	156
5.6.10 • Funções sobrecarregadas.....	158
5.7 • Declarações <i>constexpr</i> (em C++11 e em C++14).....	158
5.7.1 • Funções <i>constexpr</i> .....	158
5.7.1.1 • Em C++11.....	158
5.7.1.2 • Em C++14.....	159
5.7.2 • Objetos <i>constexpr</i> e diferença para <i>const</i> .....	159
5.8 • Lambdas (C++11).....	160
5.8.1 • Expressão lambda através de uma variável auto.....	161
5.8.2 • Capturando variáveis em lambdas.....	162
5.8.3 • Generic lambdas (C++ 14).....	162
5.9 • Tomadas de decisão (detalhamento).....	163
5.9.1 • if / else.....	163
5.9.1.1 • Cuidados a tomar com o if.....	163

5.9.1.2 • A precaução com a atribuição também se aplica aos laços.....	167
5.9.2 • switch.....	167
5.10 • Laços.....	168
5.10.1 • Porque usar laços estruturados.....	168
5.10.2 • Laço do ... while.....	169
5.10.3 • Desvios por salto incondicional.....	170
5.10.3.1 • Desvio <i>return</i> .....	170
5.10.3.2 • Desvio <i>break</i> .....	170
5.10.3.3 • Desvio <i>continue</i> .....	171
5.10.3.4 • Desvio <i>goto</i> .....	174
5.11 • Revisão do Capítulo 5.....	174
5.11.1 • Sintetizando as regras básicas para escrita de código.....	174
5.11.2 • Exercício 1.....	175
5.11.2.1 • Iniciando o exercício.....	176
5.11.3 • Exercício 2.....	179
5.11.4 • Questões para revisão.....	180

## 5.1 • Tópicos de destaque neste capítulo

### ➤ **Revendo os princípios básicos de programação e detalhando sua implementação em C e C++.**

- ◆ Linhas de instrução e operações.
- ◆ Fluxo de processamento: sequência de instruções, tomadas de decisão e desvios.

### ➤ **Mais sobre operadores.**

- ◆ Operadores compostos.
- ◆ Operadores lógicos.
- ◆ Resto de divisão.
- ◆ *Bitwise and*.
- ◆ Precedência de operadores.

### ➤ **Fluxo de processamento: funções.**

- ◆ Em C e C++, todas as linhas de instrução são escritas dentro de funções.
- ◆ Uma função é uma unidade básica de processamento e alocação de memória.
- ◆ Uma função pode chamar (colocando em execução) uma outra função qualquer.
- ◆ Funções podem (ou não) receber argumentos e retornar valores.

### ➤ **Memória: escopos.**

- ◆ Memórias podem ser reservadas dentro ou fora de funções, o que determina o seu escopo.
- ◆ O escopo de uma memória reservada determina quem poderá acessá-la (sua visibilidade) e por quanto tempo ela permanecerá reservada (seu tempo de vida).


*Tópicos de destaque, continuando...*➤ **Fluxo de processamento: sequência de instruções, tomadas de decisão e desvios.**

- ◆ Uma linha de instrução é composta por uma ou mais operações, onde cada operação depende de um operador que pode ter uma precedência superior a outros.
- ◆ Se nada de especial ocorrer, as linhas de instrução serão executadas linearmente, uma após a outra.
- ◆ A execução sequencial das instruções será quebrada quando houver uma decisão a tomar (por exemplo, quando comparamos os valores armazenados em duas diferentes áreas de memória).
- ◆ Comparações (ou, em geral, a avaliação de uma expressão) representam tomadas de decisão, que implicam em desvios no fluxo de processamento, levando a caminhos diferentes.
- ◆ Veremos com mais detalhe as tomadas de decisão com [ if ( <condição> ) ... else ... ].

➤ **Fluxo de processamento: laços.**

- ◆ Quando é necessário repetir a execução de uma sequência de instruções por um certo número de vezes, precisamos que o fluxo de processamento volte ao início dessa sequência até que uma determinada condição seja satisfeita, e a repetição deva ser encerrada.
- ◆ Essas repetições são denominadas laços.  
Em C e C++ temos algumas palavras reservadas que permitem criar laços de modo muito simples.  
Além do while e do for, analisados no capítulo anterior, veremos todos os controles de laço e alguns de seus detalhes.

➤ **Protótipo de funções e arquivos *header*.**

 Como já vimos, o fluxo de processamento é baseado em **sequência de instruções, tomadas de decisão e desvios**.

## 5.2 • Linhas de instrução; declarações e operações

**Linhas de instrução** são constituídas por **declarações** e **operações**.

- As **declarações** representam pedidos de **reserva de memória** para um determinado **tipo de dados**.
- As **operações** envolvem **operadores** e **operandos** (ou *argumentos da operação*) e executam uma determinada computação.
- Tanto declarações como operações podem ser **encadeadas** de diferentes modos em **uma mesma linha de instrução**.

**Exemplos com declarações e operações únicas ou encadeadas:**

Declarando memórias do mesmo tipo <i>em uma única linha de instrução</i> :	<b>Idem. Mas inicializa 'a', ao atribuir-lhe um valor na própria linha de sua declaração:</b>	Declarando memórias <i>em diferentes linhas de instrução</i> :
↓	↓	↓
Declarações: int a , b , c , d ;	Declarações: int <u>a = 10</u> , b , c , d ; // <i>inicializou 'a'...</i>	Declarações: int <u>a = 10</u> ; // <i>inicializa</i> int b ; int c ; int d ;
↓	↓	↓
Operações: a = 10 ; // <b>uma operação</b> b = a ; // <i>idem</i> c = a + b ; // <b>duas</b> d = a + b - c ; // <b>três</b>	Operações: b = a ; c = a + b ; d = a + b - c ;	Operações:  // <i>as mesmas</i> // <i>do quadro à esquerda</i>

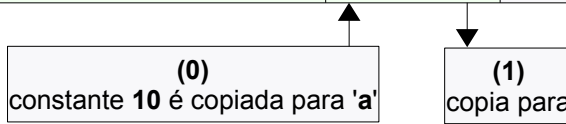
Todos os exemplos acima implicam em pedidos de reserva **de memória**, seguidos de **operações** cuja natureza é revelada pelos seus **operadores** (os símbolos das operações), os quais usam essas memórias como argumentos ou (operandos).

- Ou seja, a memória é constituída por um conjunto de posições, cada qual representada por um endereço, do mesmo modo que uma rua é constituída por um conjunto de casas, cada qual com o seu número(endereço).
  - Associamos **nomes** a essas posições simplesmente como mnemônicos.
- E, uma vez declaradas as variáveis e constantes necessárias,
  - as **operações** irão simplesmente **ler** (variáveis e constantes), recuperando o seu valor;
  - e/ou **alterar** (variáveis) os valores armazenados nas posições de memória já reservadas.

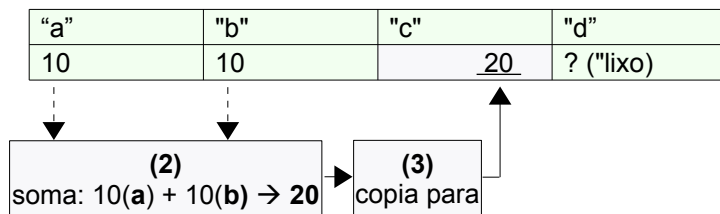
**Os exemplos da tabela acima poderiam ser visualizados assim:**

```
a = 10 ;      // (0) 10 é copiado para 'a'
b = a ;      // (1) conteúdo de 'a'(10) é copiado para 'b'
```

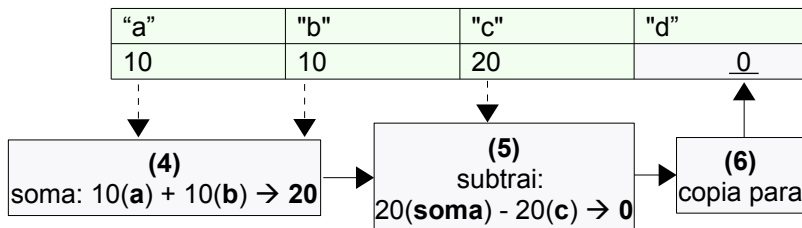
Endereços (hipótese)	1000	1004	1008	1012
Apelidos (símbolos)	"a"	"b"	"c"	"d"
Valor armazenado	10	10	? ("lixo")	? ("lixo")



```
c = a + b ; // (2) soma o conteúdo de 'a'(10) ao conteúdo de 'b'(10)
            // (3) o resultado da soma(20) é copiado para 'c'
```



```
d = a + b - c ; // (4) soma o conteúdo de 'a'(10) ao conteúdo de 'b'(10)
                // (5) subtrai o conteúdo de 'c'(20) do resultado da soma(20);
                // (6) o resultado da subtração(0) é copiado para 'd'
```



**Operações** (ou expressões) são a base para a criação de linhas de instrução, e, desse modo, para a criação de programas de computador. E operações dependem de **operadores**.

## 5.3 • Operadores

### 5.3.1 • Acrescentando mais alguns operadores

Acima, já vimos algumas operações e seus operadores. Para os próximos exemplos e exercícios precisaremos de novos operadores, conforme a **tabela abaixo**:



Mais alguns operadores		
<b>Operadores aritméticos:</b>		
%	Resto de divisão entre inteiros (módulo)	[ x % y ]; retorna o resto da divisão de 'x' por 'y'
<b>Operadores aritméticos para operações executadas <i>bit a bit</i> (<i>bitwise</i>):</b>		
&	<u>and</u>	[ x & y ]; operação <u>and</u> , executada <i>bit a bit</i> entre 'x' e 'y'
<b>Operadores compostos (operações aritméticas seguidas de atribuição)</b>		
+=	soma e atribui	[ x += y ]; soma 'x' a 'y' e armazena resultado em 'x'; equivale a [ x = x + y ];
-=	subtrai e atribui	[ x -= y ]; subtrai 'y' de 'x' e armazena resultado em 'x'; equivale a [ x = x - y ];
*=	multiplica e atribui	[ x *= y ]; multiplica 'x' por 'y' e armazena resultado em 'x'; equivale a [ x = x * y ];
/=	divide e atribui	[ x /= y ]; divide 'x' por 'y' e armazena resultado em 'x'; equivale a [ x = x / y ];
%=	retorna resto da divisão e atribui	[ x %= y ]; armazena o resto da divisão de 'x' por 'y' em 'x'; equivale a [ x = x % y ];
&=	<u>and</u> <i>bit a bit</i> e atribui	[ x &= y ]; <u>and</u> <i>bit a bit</i> entre 'x' e 'y'; armazena resultado em 'x'; equivale a [ x = x & y ];
++	incremento (+1)	[ ++x ] ou [ x++ ]; soma 1 a 'x' e armazena resultado em 'x'; equivale a [ x = x + 1 ];
--	decremento (-1)	[ --x ] ou [ x-- ]; subtrai 1 de 'x' e armazena resultado em 'x'; equivale a [ x = x - 1 ];
<b>Operadores lógicos:</b>		
&&	<u>and</u> ("e" lógico)	[ x && y ]; operação <u>and</u> lógica: 'x' e 'y' são verdadeiros?
	<u>or</u> ("ou" lógico)	[ x    y ]; operação <u>or</u> lógica: 'x' ou 'y' são verdadeiros?
!	<u>not</u> (negação)	[ !x ]; se 'x' é verdadeiro (diferente de zero) retorna falso; se 'x' é falso (igual a zero), retorna verdadeiro.
<b>Outros operadores:</b>		
( <o> )	Aumenta a <i>precedência</i> da operação entre parênteses.	

### 5.3.2 • Operador condicional ternário

Aqui temos um operador **especial**: o **operador condicional ternário**. Este operador é o único com **3 operandos** (por isso diz-se que é "ternário"):

**<condição> ? <solução verdadeira> : <solução falsa>**

Este operador trabalha como uma espécie de **if / else** simplificado, em uma única linha.

*Exemplo:*

```
int Maximo( int x, int y )
{
    return (x > y) ? x : y; // os parênteses não são necessários; apenas por clareza.
}
```

- Observe que a interrogação funciona como uma pergunta: a condição é verdadeira?
- Se for verdadeira, será considerada a expressão imediatamente após a interrogação (?), nesse exemplo uma cópia de "**x**".
- Do contrário (falsa) será considerada a expressão após os dois pontos (:), como se fosse um "**else**". Nesse exemplo, uma cópia de "**y**".



A **tabela completa** de operadores comuns a **C** e **C++** está no "**guia de consulta rápida**", seção **2**, página **479**.

### 5.3.3 • Resto de divisão

O resto da divisão entre números **inteiros** pode ser obtido de duas maneiras:

int x, y, z ; // ...	
z = x % y ;	Retorna o resto da divisão de ' <b>x</b> ' por ' <b>y</b> ', usando o operador de módulo, e o armazena em ' <b>z</b> '. Obs.: ' <b>x</b> ' e ' <b>y</b> ' podem conter quaisquer valores inteiros.
z = x & (y-1) ;	Retorna o resto da divisão de ' <b>x</b> ' por ' <b>y</b> ', usando o operador <b>and bit a bit</b> , e o armazena em ' <b>z</b> '. Obs.: ' <b>y</b> ' deve ser uma <b>potência de dois</b> . Nesse caso (e <b>apenas nele</b> ), o <b>bitwise and</b> é operado entre ' <b>x</b> ' e a <b>potência de dois menos um</b> , obtendo o resto da divisão.

**Por exemplo**, se queremos saber **se um número é par**, apuramos o resto da divisão desse número por **2**:

```
int x ;
//...
if ( x % 2 == 0 ) // se o resto da divisão de 'x' por 2 é zero:
    std::cout << "o numero " << x << " é par\n" ;
else // do contrário:
    std::cout << "o numero " << x << " é ímpar\n" ;
```

Contudo, como 2 é uma **potência de 2**, poderíamos obter esse resultado mais rapidamente usando o **and bit a bit** entre '**x**' e **1** (ou seja, 2-1):

```
if ( (x & 1) == 0 ) // se o resto da divisão de 'x' por 2 é zero:
    std::cout << "o numero " << x << " é par\n" ;
else // do contrário:
    std::cout << "o numero " << x << " é ímpar\n" ;
```

Do mesmo modo, se quiséssemos saber o resto da divisão de '**x**' por **4**, poderíamos usar o **and bit a bit** entre '**x**' e **3** (ou seja, 4-1), **já que 4 é uma potência de 2**.

```
if ( (x & 3) == 0 ) // se o resto da divisão de 'x' por 4 é zero:
    std::cout << "o numero " << x << " é divisível por 4\n" ;
```

else // do contrário:

```
std::cout << "o numero " << x << " não é divisível por 4\n" ;
```

Uma operação *bit a bit* é executada mais rapidamente (principalmente no caso de operações que envolvem **divisão** e, por decorrência **módulo**: divisão é a **mais lenta das operações aritméticas**).



Veremos as propriedades e usos das **operações bitwise** no capítulo "**referência técnica**", seção **15.3.5**, página **364**.

### 5.3.4 • Operadores compostos

Para operações aritméticas cujo resultado deve ser atribuído (armazenado) em uma variável, podemos, em alguns casos, usar os operadores compostos.

**Por exemplo:**

int x , y , z ;	Declara. // ...em seguida x, y e z receberão valores...	
x = x + 2 ;	Somar ' <b>x</b> ' e <b>2</b>	Depois armazenar o resultado <b>na própria 'x'</b>
x = x + y ;	Somar ' <b>x</b> ' e ' <b>y</b> '	Depois armazenar o resultado <b>na própria 'x'</b>
x = y + z ;	Somar ' <b>y</b> ' e ' <b>z</b> '	Depois armazenar o resultado em ' <b>x</b> '

Observe que nos dois primeiros exemplos de soma e atribuição a variável '**x**' aparece nas **duas** operações: um valor é somado a '**x**' e o resultado é armazenado na própria '**x**'. Já no terceiro exemplo, temos uma situação diferente: o que é armazenado em '**x**' é o resultado da soma de '**y**' e '**z**'. Logo, '**x**' está presente apenas na operação de atribuição, não na de soma.

**Em consequência, as duas primeiras operações poderiam ser escritas assim:**



x += 2 ; ➡	Somar primeiro (+), atribuir depois (=).	<b>Ou seja:</b> ■ some <b>2</b> a ' <b>x</b> ' ■ e depois armazene o resultado em ' <b>x</b> '
x += y ; ➡	Idem, para a soma de ' <b>y</b> ' em ' <b>x</b> '.	

E o mesmo se aplica às demais operações aritméticas (subtração, multiplicação, divisão, etc.).

Observe que o código tornou-se mais compacto e imediatamente legível: fica claro que a soma (ou outra operação aritmética) e a atribuição **envolvem uma mesma variável**. Além disso temos um caso especial para operadores compostos: **incremento e decremento**. Essas são operações muito comuns: várias vezes uma variável é afetada pela soma ou subtração do número constante **1**.

**Nesse caso poderíamos:**

x = x + 1 ;	Somar <b>1</b> a ' <b>x</b> '	Depois armazenar o resultado <b>na própria 'x'</b>
x += 1 ;	Idem, usando o operador composto.	

<code>++x ;</code> 	Idem, usando o operador de <b>incremento</b> .	
<code>x = x - 1 ;</code>	Subtrair <b>1</b> de ' <u>x</u> '	Depois armazenar o resultado <b>na própria 'x'</b>
<code>x -= 1 ;</code>	Idem, usando o operador composto.	
<code>--x ;</code> 	Idem, usando o operador de <b>decremento</b> .	



Além disso os operadores de incremento e decremento podem ser usados nas formas **pré-fixada** ou **pós-fixada**: { [ **++x** ; ] ou [ **x++** ; ] } e { [ **--x** ; ] ou [ **x--** ; ] }. Essas duas formas implicam em diferenças de resultado, podendo criar problemas dependendo do **caso de uso**. Veremos isso no capítulo "**referência técnica**", seção **15.3.4**, página **364**.

#### Detalhes:

Os operadores compostos indicam que o acesso é feito a uma única área de memória.

E no caso de incremento e decremento, além de indicarmos que a operação é feita em uma única área de memória, estamos indicando também que a soma ou a subtração envolvem sempre o número constante **1**.

Um compilador pode selecionar instruções de máquina de incremento ou decremento mais eficientes do que as instruções normais de soma e subtração, o que, obviamente, **depende de plataforma**. Por exemplo, o x86, possui as instruções **inc** e **dec** que têm justamente essa finalidade e são mais rápidas.



#### Exemplos:

Os exemplos com o laço **for** para **somar números** e para cálculo de **fatorial** foram escritos assim (página **87**, acima):


```
// a) SomaNumeros:
int inicial = 1 , final = 100 , razao = 1, result ;
for ( result = 0 ; inicial <= final ; inicial = inicial + razao
result = result + inicial ; ) // acumula para atingir a soma
```


```
// b) Fatorial:
int num = 10 , result ;
for ( result = 1 ; num > 1 ; num = num - 1 )
result = result * num ; // acumula para atingir o fatorial
```

Usando os **operadores compostos (ou ++ e --)** poderíamos escrever assim:

```
// a) SomaNumeros:
int inicial = 1 , final = 100 , razao = 1, result ;
for ( result = 0 ; inicial <= final ; inicial += razao //  +=
result += inicial //  +=
```

```
// b) Fatorial:
int num = 10 , result ;

for ( result = 1 ; num > 1 ; --num ) //  -- (decremento)

result *= num //  *=
```

### 5.3.5 • Operações lógicas

As operações lógicas tradicionais (**and** e **or**) permitem uma simplificação do código à medida em que, com elas, podemos reduzir o uso de sucessivos testes de condição individuais. E, em certos casos, são quase insubstituíveis. Além de ganhos de clareza.

#### OR (ou) lógico


Considere esta situação:

```
int funcao_qualquer()
{
    int x, y, z;
    // ...
    if ( x > y )
        return 0;
    if ( x < z )
        return 0;
    return 1;
}
```

Analisando esse código, verificamos que: [ caso 'x' seja maior que 'y' ] **ou** [ caso 'x' seja menor que 'z' ], o caminho de processamento é **o mesmo**: [ **return 0;** ].

Isso poderia ser expresso mais simplesmente com o uso do operador lógico **or**, que é simbolizado, em C e C++, por dois *pipes*: **||**

Então o código acima poderia ser escrito conforme o exemplo abaixo:

```
int funcao_qualquer()
{
    int x, y, z;
    // ...
    if ( x > y || x < z ) 
        return 0;
    return 1;
}
```

Se [ x > y ] **OU** [ x < z ]

Deixa claro que tanto uma coisa como a outra levam ao mesmo resultado.

E, consequentemente, o mesmo se aplica a múltiplas condições:

[ if ( <condição\_1> || <condição\_2> || ... || <condição\_n> ) → <único\_caminho> ].

#### AND (e) lógico

Em outras situações, ao contrário, é preciso que **duas (ou mais) condições sejam verdadeiras** para que um determinado caminho seja seguido.

Ou seja: nesse caso temos uma conexão **and** (duas ou mais condições têm que ser verdadeiras), que, em C e C++, é simbolizada, com dois "e-comerciais": **&&**.

Considere esta situação:

```
int funcao_qualquer()
{
    int x, y, z;    // ...
    if ( x > y )
    {
        if ( x < z )
            return 0;
    }
    return 1;
}
```

Analisando esse código, verificamos que: [ caso 'x' seja maior que 'y' ] e [ caso 'x' seja menor que 'z' ], um determinado caminho de processamento será seguido: [ **return 0;** ]. Ou seja: apenas se as duas condições forem verdadeiras é que teremos um desvio do fluxo para esse caminho. Desse modo, o código acima poderia ser escrito conforme o exemplo abaixo, **deixando claro que apenas se duas(ou mais) condições forem verdadeiras é que teremos o mesmo resultado:**

```
int funcao_qualquer()
{
    int x, y, z;    // ...
    if ( x > y && x < z )
        return 0;
    return 1;
}
```

Se [  $x > y$  ] E [  $x < z$  ]  
**Deixa claro** que somente se as **duas** condições forem **verdadeiras** o caminho a seguir será o mesmo.

## Not (negação) lógico

O operador **not** (simbolizado por um sinal de exclamação) transforma um valor **verdadeiro em falso e vice-versa**. Exemplos com tipos inteiros e com o **bool**:

int i = 100 ;	i = !i ;	Inicialmente " <b>i</b> " é verdadeira (diferente de zero). Após o <b>not</b> (!) torna-se falsa (valor zero, <b>false em C++</b> ).
int i = 0 ;	i = !i ;	Inicialmente " <b>i</b> " é falsa (zero). Após o <b>not</b> (!) torna-se verdadeira (valor um, <b>true em C++</b> ).
bool b = true ;	b = !b ;	Inicialmente " <b>b</b> " é verdadeira ( <b>true</b> ). Após o <b>not</b> (!) torna-se falsa (valor <b>false</b> ).
bool b = false ;	b = !b ;	Inicialmente " <b>b</b> " é falsa ( <b>false</b> ). Após o <b>not</b> (!) torna-se verdadeira (valor <b>true</b> ).

## Outros exemplos com operações relacionais e lógicas:

int x, y, z ; bool r ;	// ... Seguem-se operações que alteram " <b>x</b> ", " <b>y</b> " e " <b>z</b> "...
r = x > y ;	Se [ " <b>x</b> " for maior-que " <b>y</b> " ], " <b>r</b> " receberá o valor <b>true</b> . Do contrário, <b>false</b> .
r = x    y ;	Se [ " <b>x</b> " ] <b>ou</b> [ " <b>y</b> " ] forem diferentes de zero, " <b>r</b> " receberá o valor <b>true</b> . Do contrário, <b>false</b> .
r = x && y ;	Se [ " <b>x</b> " ] <b>e</b> [ " <b>y</b> " ] forem diferentes de zero, " <b>r</b> " receberá o valor <b>true</b> . Do contrário, <b>false</b> .
r = x > y    x < z ;	Se [ " <b>x</b> " for maior-que " <b>y</b> " ] <b>ou</b> [ " <b>x</b> " for menor-que " <b>z</b> " ], " <b>r</b> " receberá o valor <b>true</b> . Do contrário, <b>false</b> .
r = x > y && x < z ;	Se [ " <b>x</b> " for maior-que " <b>y</b> " ] <b>e</b> [ " <b>x</b> " for menor-que " <b>z</b> " ], " <b>r</b> " receberá o valor <b>true</b> . Do contrário, <b>false</b> .

```
r = ! (x > y) ;
```

Se [ "**x**" for maior-que "**y**" ], o resultado da comparação será **true**. Do contrário, **false**.  
Mas o **not** inverte esse resultado. Desse modo, "**r**" receberá **false** se [ "**x**" for maior-que "**y**" ]. Do contrário, **true**.

### 5.3.6 • Em conclusão

Uma operação é identificada por seu **operador**. Até agora vimos:

- O operador de **atribuição**.
- O operador de **chamada de função**.
- Os operadores **aritméticos** (inclusive o operador de módulo e o *bitwise and*).
- Os operadores **relacionais**.
- Os operadores **compostos** (inclusive incremento e decremento).
- Os operadores **lógicos and, or e not**.
- o operador **condicional** ternário.

Além disso, precisamos saber que existem regras de **precedência** para a execução de operadores, as quais visam a disciplinar **operações encadeadas**.

## 5.4 • Precedência de operadores.

Em operações encadeadas, algumas operações são realizadas anteriormente a outras, mesmo que estejam em uma posição posterior.

**Exemplo:**

```
int x = 10 , y = 20 , z = 30 , r ;
```

```
r = x + y * z ; // a multiplicação é executada antes da soma...
```

**A linha de instrução acima será executada na seguinte ordem:**

- a. a **multiplicação** ("**y \* z**") será executada primeiro;
- b. em seguida, o **resultado da multiplicação** será **somado** a "**x**"
- c. finalmente, o **resultado da soma** será **atribuído** a "**r**" (ou, dito de outro modo, **copiado** para "**r**")

Isso é executado assim porque a **multiplicação** tem uma precedência **mais alta** do que a **soma** e esta, por sua vez, tem precedência **mais alta** do que a **atribuição**, representada pelo operador "=" (também chamado *operador de cópia*).

Poderíamos **mudar essa ordem**, com o operador "<operação>", que **eleva a precedência** de uma operação colocada entre **parênteses**. Se a finalidade desse cálculo exigisse que a soma fosse realizada em primeiro lugar, bastaria fazer:

```
r = (x + y) * z ; // a soma é executada antes da multiplicação...
```

**Nessa forma, as operações serão executadas na seguinte ordem:**

- a. a **soma** ("**x + y**") será executada primeiro;
- b. em seguida, o **resultado da soma** será **multiplicado** por "**z**"
- c. finalmente, o **resultado da multiplicação** será **atribuído** a "**r**".



A **tabela de precedência de operadores** está no “**guia de consulta rápida**”, seção **3**, página **481**. Nessa tabela, além dos operadores comuns a **C** e **C++**, veremos também **operadores que só existem em C++**.

Informações, conceitos e detalhes sobre **operações** e operadores podem ser encontrados no capítulo “**referência técnica**”, seção **Error: Reference source not found**, página **Error: Reference source not found**.



Considere também que **explicitar** a precedência de operações com o operador “( **<operação>** )”, pode melhorar a **legibilidade** do código em linhas de instrução com  **muitas** operações encadeadas.

#### Exemplo:

```
a = b + c * d / e * f + g - h ;
```

Digamos que essas operações, em função de sua **finalidade**, já estejam aí **corretamente** posicionadas, devido às **regras de precedência**.

Contudo, ao ler essa linha, perdemos um pouquinho de tempo, pois precisamos analisar as precedências envolvidas para ter certeza de que está tudo realmente correto.

**Essa linha seria mais rapidamente legível se a escrita fosse mais explícita:**

```
a = b + ( ( c*d ) / ( e *f ) ) + ( g - h ) ;
```

## 5.5 • Finalização de uma linha de instrução.

Como já pudemos observar nos exemplos de código fonte acima (e nas regras básicas de gramática), linhas de instrução são **encerradas** com um “**;**” (**ponto e vírgula**).

Essa é uma regra básica de **C** e **C++**. Isso significa que a **linha de texto** é uma coisa e a **linha de instrução** é outra coisa.



**Linhas de instrução** são escritas em editores de texto, formando linhas de texto, **mas não se confundem com estas**.

#### Exemplo:

A linha de instrução abaixo

```
a = b ;
```

poderia ser escrita assim

```
a
=
b
;
```

Embora a segunda forma pareça (e de fato é!) uma escrita mais **confusa**, ela está sintaticamente correta. No caso, temos **uma única linha de instrução em quatro linhas de texto**. Pois o que termina uma linha de instrução é o ponto e vírgula.

## 5.6 • Funções.

Acima vimos:

- alocação (reserva) de **memória**;
- **linhas de instrução** contendo **declarações** e/ou **operações**;
- **tomadas de decisão** (if ... else...).



E já vimos também onde escrever tudo isso: declarações e instruções devem estar **dentro de uma função**.

- Uma função é o elemento mais importante do controle de fluxo, porque representa a **unidade básica** para a escrita de código, permitindo que um determinado trecho de código **desvie a execução** para outro trecho (isto é, uma outra função) ordenadamente.
- Assim, escrevemos funções que executam apenas uma pequena tarefa.
- E, quando uma função precisa do serviço prestado por outra, simplesmente **chama** essa outra função, transferindo o fluxo de processamento para esta.
- Ao encerrar sua execução, a função retorna ao ponto onde foi chamada (**retorna para quem a chamou**).

<b>Exemplo: função "Mínimo"</b> - recebe dois valores inteiros e retorna o menor deles. Pode ser chamada assim: <b>int x, y, z ;</b> //... <b>z = Mínimo ( x, y ) ;</b>	<pre>int Minimo ( int a, int b ) {     if ( a &lt; b )         <b>return a ;</b>     else         <b>return b ;</b> }</pre>
---	---

Em uma outra função qualquer, quando quisermos saber qual é o menor de dois valores, simplesmente **chamamos** a função *Mínimo*:

 A linha abaixo, irá **transferir o processamento** para a função '*Mínimo*' que executará suas instruções e, **ao final, retornará ao ponto de sua chamada**:

```
z = Minimo ( x, y ) ;           // o resultado retornado por 'Minimo'
                               // será copiado para 'z'
```

### 5.6.1 • Parâmetros e valor de retorno de uma função.

Observe, no próprio exemplo da função *Mínimo*, que uma função pode receber **argumentos** (passados por quem a chama) e armazená-los em variáveis denominadas **parâmetros**, bem como **retornar um valor**, que representa o seu resultado formal e que pode ser memorizado por quem a chamou para uso futuro.

#### Parâmetros.

Argumentos são objetos, passados como argumentos por quem chama a função. Servem para que ela tenha informações, visando a atingir um resultado de acordo com sua finalidade. Eles são recebidos pela função em memórias (variáveis) designadas como **parâmetros da função**. No exemplo da função *Mínimo*, esses parâmetros são as variáveis "**a**" e "**b**".

**Assim sendo, de acordo com o exemplo acima:**

- a função *Mínimo* tem dois **parâmetros** do tipo **int**;
  - logo deve ser **chamada com a passagem de dois argumentos do mesmo tipo**, ou teremos um erro de compilação, pois a função **precisa** desses dois argumentos para cumprir sua **finalidade**.

#### Valor de retorno.

Além disso essa função **retorna** um valor o qual representa o **resultado** do seu trabalho. Esse resultado é um **valor** de determinado **tipo**.

Acima, no exemplo de **chamada** da função *Minimo*, `z = Minimo ( x, y );` podemos perceber que ela retorna um valor e também que esse valor pode ser recuperado e memorizado por quem chamou a função.

✎ Em conclusão, uma função é caracterizada por um **valor de retorno**, um **nome** e uma **lista formal de parâmetros**, a qual é definida pelos parênteses após o nome.

E o exemplo da função <b>Minimo</b> permite-nos descrever <b>todos os elementos</b> que fazem parte de uma <b>função</b> :	Valor de retorno	Nome	Parâmetros
	int	Minimo	( int a , int b )

### 5.6.2 • Funções sem parâmetros e/ou valor de retorno.

✎ **Não necessariamente** uma função precisa **receber argumentos ou oferecer retornos**. Tudo isso **depende**, exclusivamente, **da sua finalidade**.

- Contudo, isso **não altera a sua sintaxe**. Ela **continua tendo** uma indicação de **retorno** (que pode ser "*retorno vazio*") e uma **lista formal de parâmetros** entre os **parênteses**. A lista poderá estar vazia ou não, mas a regra geral é a mesma.

**Exemplos:**

#### a. Função que não retorna valor.

**Função Imprime;** imprime um número inteiro e **não retorna** um valor:

```
U // "void" (ou "vazio") indica que a função não retorna um valor...
void Imprime( int n )
{
    // ... instruções necessárias para a impressão do número 'n' ...
}
```

✎ Se o retorno da função é definido como **void** (vazio), isso significa que ela **não retorna valor**.

#### b. Função que não recebe argumentos.

**Função isPrinterOn:** **não recebe argumentos** pois sua **finalidade** é descobrir se há uma impressora ligada e ela **não precisa** de informações de entrada para descobrir isso. Retorna verdadeiro ou falso, dependendo da situação. Como no **exemplo abaixo**:

```
bool isPrinterOn ( )
{
    bool on;
    // ...instruções necessárias para descobrir se a impressora está ligada...
    // a variável "on" assumirá "true" em caso afirmativo - ou "false".
    return on;
}
```

#### c. Função que não retorna valor e não recebe argumentos.

**Função `exit_on_error`:** verifica se ocorreu algum **erro** durante a execução. Caso tenha ocorrido, chama a função **"exit"** da biblioteca **C** para **encerrar o programa**. Não recebe argumentos, pois as informações de que necessita estão em um indicador de erros (variável global **"errno"**) fornecida pela própria biblioteca. Também não retorna valor, já que simplesmente ou encerra o programa ou não executa nada.

```
void exit_on_error()
{
    // a variável "errno" da biblioteca C informa o último erro ocorrido;
    // se contiver zero é porque não existiram erros.
    if ( errno != 0 ) // diferente de zero? então ocorreu um erro...
    {
        std::cout << "Este programa fez algo errado\n" ;
        std::cout << "Encerrando agora\n" ;
        exit( -1 ) ; // encerra o programa acusando erro (retorno diferente de zero)
    }
}
```

### 5.6.3 • A função `main`

Como vimos, um programa em **C** ou **C++** tem suas linhas de instrução sempre aninhadas em funções.

E vimos também que para que um programa seja iniciado é preciso que exista uma **função privilegiada** que cumpra o papel de **ponto de entrada** de uma aplicação executável. E essa é a função que deve ter um nome especial: **"main"** - por **convenção** da linguagem.

✎ Um programa executável deve, **obrigatoriamente, conter uma e somente uma função** com esse nome especial (**main**). A execução inicia aí.

Escrevemos nossas funções em **arquivos** (formalmente denominados como **unidades de tradução** ou **módulos**). Os nomes desses arquivos, normalmente (mas não necessariamente), têm a extensão **".c"** (Linguagem **C**) ou **".cpp"** (Linguagem **C++**). E a função **main** poderá estar em **qualquer um** dos arquivos envolvidos em um projeto.

✎ **Não importa** em qual **arquivo** ou em qual **posição** dentro de um arquivo esteja escrita a função **main**. Independentemente disso, ela será sempre o ponto de entrada, dando início à execução do programa.

E dentro de **main**, isto é, em suas linhas de instrução, outras funções podem ser chamadas (*por exemplo*, para executar um *menu* de opções disponíveis para seleção do usuário), dando assim seguimento ao programa.

Acima, já vimos vários exemplos de implementação da função **main**.

✎ E, lembrando, quando **main** retorna, o programa é **encerrado**.

Agora é o momento de **acrescentar mais algumas informações**. Temos usado a função **main** retornando um inteiro. Mas para que serve esse retorno? Além disso, **main** também pode **receber argumentos**.

### 5.6.3.1 • Retorno de main.

Em todos os exemplos onde a função **main** já apareceu, temos o seguinte modelo:

```
int main()
{
    return 0;
}
```



Este modelo (**main retorna int**) está em **conformidade com o padrão** da linguagem (veremos abaixo porque).

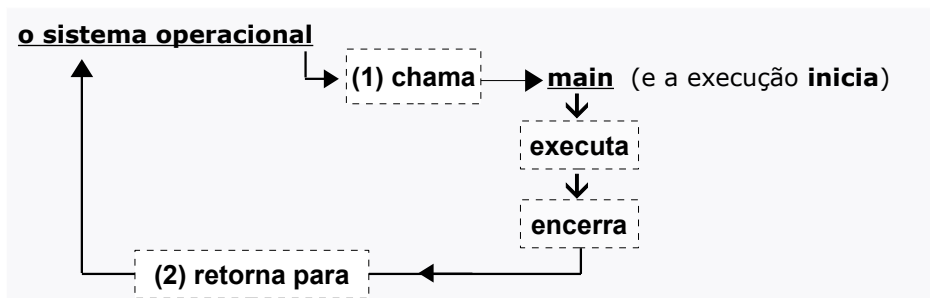
Mas talvez você encontre, em algum lugar, exemplos de **main** com o seguinte modelo:

```
void main()
{
}
}
```



Este modelo ("**void main**", ou seja: não retorna valor) está em **desacordo com o padrão** da linguagem.

No primeiro exemplo desta apostila ("programa mínimo", seção **2.2.1**, página **41**, acima) foi exibido o seguinte diagrama que mostra como **main** começa e encerra: ela é chamada pelo sistema operacional (e, assim, o próprio programa tem início) e, ao final, retorna para quem a chamou (o próprio SO):



Na realidade **main** não é chamada diretamente pelo SO. Há um pequeno trecho de código, adicionado pelo compilador, que permite que o sistema coloque a aplicação em execução - e é nesse código adicional que **main** é chamada.

Mas esse detalhe de implementação não muda o que é sintetizado no diagrama acima: para todos os efeitos, **naquilo que é visível no código fonte**, o programa inicia em **main**. E, como qualquer função, **main** retorna para quem a chamou, que, no seu caso, é o sistema operacional.

**A pergunta é: o sistema operacional espera algum valor de retorno?**

Isso **depende** do sistema operacional que esteja em uso. **Alguns** sistemas operacionais (*Unix/Linux*, por exemplo) consideram esse valor de retorno (**zero**) como uma indicação de que o programa encerrou **sem erros**.

Dito de outro modo, se um programa é interrompido porque tentou realizar uma operação ilegal, um código de erro (diferente de zero) será atribuído como o seu valor de retorno.

Nesse caso, o programa não teve a chance de concluir e não atingiu o "**return 0**" que está no final de **main**. Mas, se atingir esse ponto, estará indicando um "erro zero" isto é, ausência de erro.

Existem outros sistemas que **desconsideram** o retorno de **main**. Mas, sendo multi-plataforma, a linguagem deve satisfazer a todos - e por isso **main** deve ter um retorno **int** e retornar **zero** (exceto se quiser acusar deliberadamente uma condição de erro, retornando qualquer outro valor do tipo **int**).

### 5.6.3.2 • Parâmetros de **main**.

A função **main** pode ser implementada **com ou sem parâmetros** - isso porque há programas que **esperam receber argumentos** (passados na linha de comando, ou por algum outro meio) e também há programas que **não têm nada a fazer com argumentos** que eventualmente lhes sejam passados (provavelmente por engano do usuário, já que o programa não os espera).

**Então a função main pode ser implementada de duas formas, ambas corretas:**

sem parâmetros	com parâmetros
<pre>int main() {     return 0; }</pre>	<pre>int main( int argc , char *argv[] ) {     return 0; }</pre>

O primeiro parâmetro, "**argc**", é um inteiro que informa a quantidade de argumentos que foram passados para o programa e que serão recebidos no segundo parâmetro ("**argv**"). Por enquanto, ainda não sabemos identificar o tipo do segundo parâmetro ("**char \* argv[]**"). Esse tipo será visto adiante (seção **15.5.3.5**, página **412**). Mas, por enquanto, podemos assumir que se trata de uma **lista** (vetor) **de strings**.

**E é fácil perceber como funcionaria o tratamento desses parâmetros com um pequeno exemplo:**

a. Dentro do diretório-base de exercícios ("**cursoCPP**"), crie um novo diretório para este exemplo; por exemplo, "**05\_main\_arg**". No editor de textos, digite o código abaixo e o salve como "**main.cpp**".

- Desse modo teremos algo como: "**cursoCPP/05\_main\_arg/main.cpp**"

b. **Código fonte:**

```
#include <iostream>
int main( int argc , char *argv[] )
{
    int a ;
    // percorre a lista de argumentos (o total deles é "argc")
    for ( a = 0 ; a < argc ; ++a )
        std::cout << argv [ a ] << "\n" ; // imprime um argumento
        // - ou seja, uma string em estilo "C" da lista de strings "argv"
    return 0;
}
```

c. **Compilar.**

d. **Executar** o programa, **passando dois argumentos: "testando" e "argumentos"**. Se o nome do executável for **05\_main\_arg**, teremos:

**Em Windows:**                    **05\_main\_arg testando argumentos**  
**Em Unix/Linux:**            **./05\_main\_arg testando argumentos**

Como **resultado** da execução, será impresso no monitor de vídeo:

**05\_main\_arg** [ no *Windows* teremos o ".exe" ]  
**testando**  
**argumentos**  
 > [ *próxima posição do cursor no monitor de vídeo* ]

Como podemos ver, o **primeiro** argumento **sempre** existe: ele é o próprio nome do executável ("**05\_main\_arg**" ou "**05\_main\_arg.exe**", no caso do *Windows*).

O segundo e terceiro argumentos ("**testando**" e "**argumentos**") são aqueles que foram passados explicitamente na linha de comando ao executarmos o programa.

Há uma série de programas que tratam os argumentos passados na linha de comando. E também há muitos que simplesmente desprezam o que quer que seja passado.

### 5.6.4 • Protótipos de funções.

Quando uma função chama outra, a **função chamada** deve ser **conhecida** pelo compilador. Esta tanto pode estar no mesmo arquivo da chamadora como também em outro arquivo qualquer. Além disso, se estiver no mesmo arquivo, poderá estar escrita acima ou abaixo da chamadora. O **compilador** não olha para baixo nem para os lados... Quando ele analisa se um determinado símbolo está sendo usado de forma correta, **ele só olha para cima**.

#### Exemplo:

```
#include <iostream>
// aqui está escrita a função "Fatorial":
int fatorial( int num ) { /* ... */ }
int main ( )
{
  std::cout << "fatorial de 5 = " << Fatorial ( 5 ) << "\n";
```



**OK:** o compilador encontra "Fatorial" **acima**.

```
std::cout << "soma dos números entre 1 e 10 = " <<
somaNumeros( 1, 10 ) << "\n";
```



**Erro:** o compilador não encontra a função "somaNumeros", pois ela está escrita **abaixo**.

```
}
// aqui está escrita a função "somaNumeros":
int somaNumeros( int inicial, int final ) { /* ... */ }
```

Por isso, se uma função chamada estiver escrita abaixo da chamadora (ou até mesmo em outro arquivo), **será preciso instruir o compilador sobre sua sintaxe**, isto é, **como essa função deve ser chamada**, obedecendo a seu valor de retorno e a seus parâmetros.

Para isso escrevemos **protótipos** de funções, indicando os seus **nomes**, **valores de retorno** e **suas listas de parâmetros** (em outras palavras, determinamos sua **sintaxe**).

**Exemplo:**

```
void Func_1 ( ) ;      // protótipo da função "Func_1"
void Func_2 (int x) ;  // protótipo da função "Func_2"
int  Func_3 ( ) ;     // protótipo da função "Func_3"
```



Acima, declaramos que, em algum lugar, existirão funções com **esses nomes, esses valores de retorno e essas listas de parâmetros.**

Agora o **compilador** poderá **verificar** se elas estão sendo **chamadas corretamente ou não.**

```
int main()
{
    // chama a função "Func_1":
    Func_1() ; // correto: foi chamada de acordo com seu protótipo.
    int a = Func_1() ; //  Erro: tentando copiar retorno "void"...
    Func_1( 5 ) ;      //  Erro: essa função não recebe argumentos

    // chama a função "Func_2":
    Func_2( 5 ) ; // correto: foi chamada de acordo com seu protótipo.
    int b = Func_2( 5 ) ; //  Erro: tentando copiar retorno "void"...
    Func_1() ;      //  Erro: essa função deve receber um argumento int.

    // chama a função "Func_3":
    Func_3 ( ) ; // correto: foi chamada de acordo com seu protótipo;
                  // e não é obrigatório copiar o seu valor de retorno.
    int c = Func_3 ( ) ; // correto, de acordo com o protótipo.
    int c = Func_3( 5 ) ; //  Erro: essa função não recebe argumentos.

    return 0 ;
}

// "Func_1" está implementada aqui:
void Func_1() { /* .... */ }

// "Func_2" e "Func_3" estão implementadas em outros arquivos.
```

**5.6.5 • Arquivos header**

No exemplo acima, vimos os seguintes **protótipos** de funções:

```
void Func_1 ( ) ;      // protótipo da função "Func_1"
void Func_2 (int x) ;  // protótipo da função "Func_2"
int  Func_3 ( ) ;     // protótipo da função "Func_3"
```

E, no final, é afirmado que "**Func\_1**" está implementada **nesse mesmo arquivo**, abaixo da função **main**. Já "**Func\_2**" e "**Func\_3**" estão implementadas em **outros arquivos**. Nos dois casos, os protótipos permitiram que o compilador pudesse entender que símbolos são esses e como podem ser usados. Isto é: qual é a **regra de sintaxe** a ser seguida em seu uso (nome, argumentos e retorno cabíveis). Isso permite que possamos implementar as funções em qualquer lugar, independentemente de onde é chamada.

Mais sobre o papel do compilador e do *link-editor*: **anexo A**, página **422**.

Podem existir muitos arquivos fonte em uma mesma aplicação, e se ocorrerem chamadas a essas funções em mais do que um arquivo, teríamos que **copiar e, em seguida, colar esses protótipos** em todos os arquivos onde existissem chamadas a elas:

arquivo_1.cpp	arquivo_2.cpp
<pre>int Func_3 ( ) ; // protótipo void funcao_do_arquivo_1 ( ) {     Func_3( ) ; // chama "Func_3" ; }</pre>	<pre>int Func_3 ( ) ; // o <b>mesmo</b> protótipo void funcao_do_arquivo_2 ( ) {     Func_3( ) ; // chama "Func_3" <u>também</u> }</pre>

Isso não seria necessário se tivéssemos colocado esses protótipos em um único lugar (um único arquivo). Esse arquivo seria então incluído em todos os arquivos fonte onde existissem chamadas a essas funções.

É por isso que escrevemos [ **#include <iostream>** ] no topo de todos os fontes que usamos até aqui. E isso foi feito devido ao uso de **cout**, que não é uma palavra reservada mas sim um recurso declarado no arquivo **iostream**. Nesse arquivo encontramos:

```
ostream cout ; // "cout" é um objeto do tipo "ostream"
```

O que permite que o compilador saiba o que é e o que não é possível fazer com **cout**.

Podemos usar o **mesmo método**. Digamos que, no exemplo acima, criássemos um arquivo denominado "**funcoes.h**", sendo ".h" a abreviatura de *header*, isto é, algo que é incluído no topo ou **previamente** ao uso. A extensão ".h" é usual, mas não é uma regra da linguagem: não existe uma regra para nomes e extensões de arquivos.

funcoes. h	arquivo_1.cpp	arquivo_2.cpp
<pre>void Func_1 ( ) ; // protótipo void Func_2 (int x) ; // idem int Func_3 ( ) ; // idem  void f_a_1(); // idem void f_a_2(); // idem  extern int g_var; // <b>U</b> // <b>declara</b> que "g_var" existirá // (será <b>definida</b>) em <b>algum</b> // arquivo fonte. // E, sendo global, poderá // ser usada em <b>qualquer</b> função // de <b>qualquer</b> arquivo // da aplicação // (<b>escopo da aplicação</b>).</pre>	<pre><b>#include "funcoes.h"</b> // <b>declara</b> e // <b>define</b> g_var: int g_var = 0; // implementa f_a_1: void f_a_1 ( ) {     Func_1( ) ;     Func_2( 10 );     Func_3( ) ;      // <b>acessa</b> g_var:     <b>g_var = 5;</b>     // ..... } // <b>func_1</b> e <b>func_2</b> são // implementadas neste // arquivo: void func_1() {     // ..... } void func_2(int x) {     // ..... }</pre>	<pre><b>#include "funcoes.h"</b> // <b>inicia</b> em main: int main() {     // .....     <b>func_a_1();</b>     <b>func_a_2();</b>     // ..... } // <b>implementa</b> f_a_2 void f_a_2 ( ) {     Func_1( ) ;     Func_2( 20 );     Func_3( ) ;      // <b>acessa</b> g_var:     <b>g_var = 20;</b>     // ..... } // <b>func_3</b> é // implementada neste // arquivo: int func_3() {     // ..... }</pre>

Naturalmente, se uma função é usada em **apenas um** arquivo fonte não seria necessário inserir seu protótipo em um *header*. Bastaria que ele existisse nesse único arquivo.



### 5.6.6 • Escopo de uma função e escopo global.

Vimos que uma função é uma **unidade de execução** no fluxo geral de processamento de um programa, o que significa que ela executa uma **tarefa bem definida**. Assim, um programa nada mais é do que a articulação de diversas tarefas executadas através de **chamadas de função**, as quais trabalham com determinados **dados** (memória).



E uma função, em **C** e **C++**, além de ser uma **unidade de execução de instruções**, também é uma **unidade de alocação de memória**.

Como já vimos no capítulo 4, uma função (e qualquer **bloco interno** a uma função) pode ter **memórias próprias**, que nenhuma outra função ou bloco poderá acessar. Pois tais memórias têm um **proprietário**: o bloco que as declarou. Por isso dizemos que o seu **escopo é local**.

Além disso, podemos declarar memórias **fora de qualquer função**, o que significa que essas memórias não pertencem a nenhuma função em particular (**não têm um proprietário**). Logo, são **públicas ou globais**, podendo ser acessadas por qualquer uma das funções de uma aplicação.

**Exemplo:**

```
int g_varGlobal;
```



A variável "**g\_varGlobal**" foi declarada **fora de qualquer função**. Ou seja: **não tem proprietário**. Então é pública... **Escopo global** ou **escopo da aplicação**.

```
void Funcao_1 ( int x ) ; // protótipo...
```

```
int main()
{
```

```
    int x; // Declarada dentro de main; pertence a main. Escopo local.
```



A variável "**x**" foi declarada em "**main**". Logo, **pertence a "main"**, **não** podendo ser acessada **por qualquer outra função**. **Escopo local**.

```
// ...
```

```
Funcao_1 ( x ) ; // Chama a função "Funcao_1", passando uma cópia de "x".
```

```
g_VarGlobal = 1; // Acessa a variável global.
```

```
return 0 ;
```

```
}
```

```
void Funcao_1 ( int x )
```

```
{
```

```
    int a;
```



Tanto "**x**" (**parâmetro**) como "**a**" foram declaradas em "**Funcao\_1**".

Logo, **pertencem a "Funcao\_1"**, **não** podendo ser acessadas por qualquer outra função. **Escopo local**.

```
g_VarGlobal = 2; // Também aqui podemos acessar a variável global
```

```
}
```

### 5.6.7 • Chamadas de função e seu custo.

O **controle de fluxo** de processamento mais importante da linguagem é baseado na organização de uma aplicação em diversas sub-rotinas (**funções**). Assim a **chamada de função** representa um desvio de fluxo organizado (estruturado) já que a operação de salto é feita com garantia de retorno ao ponto imediatamente seguinte à sua chamada.


---

✎ Contudo, a **simples chamada de uma função** tem um **custo** (independentemente do que a função faz em seu corpo, isto é do custo de sua execução).

---

Por exemplo, em:

```
int z = maximo(a, b);
// ...
int maximo( int x, int y ) { return x>y ? x : y; }
```



Vejam os custo decorrente dessa chamada de função:

- Para **cada** argumento que é passado (para satisfazer um parâmetro da função) é necessária **uma instrução de máquina**. Assim para uma função com dois parâmetros (como é o caso de "maximo") são necessárias duas instruções de máquina **apenas** para a passagem de argumentos.
- Em seguida, é necessária **uma** instrução de máquina para chamar a função.
- Se a função retorna um valor, esse valor deve ser **armazenado** de algum modo enquanto a chamada de função ainda estiver ativa.
- Se a função recebe parâmetros, eles serão, um a um, armazenados na **pilha**. Quando a função retorna é preciso que o apontador de pilha retroceda para o ponto em que estava antes da chamada ("liberando" essas memórias).

### 5.6.8 • Funções inline.

Funções **inline** são escritas como uma função qualquer, mas **possivelmente não** serão executadas como uma função, evitando o custo da simples chamada de função.

Pois o compilador **poderá** simplesmente **trocar a chamada** da função **pelo código** da própria função em todos os lugares em que ela estiver sendo chamada.

Melhor dizendo: o compilador **avalia** o que é melhor fazer (melhor relação "custo/benefício").

- Assim, se a função estiver sendo chamada em **muitos lugares**, mas for **pequena** (contendo poucas instruções) o compilador, **possivelmente**, irá trocar as chamadas pelo código da função. No executável, portanto, a função **provavelmente não** existirá.
- Se a função for **grande** (contendo muitas instruções) mas estiver sendo chamada em **um único lugar** o compilador também **poderá** efetuar a troca, pois de qualquer modo essa quantidade de linhas de instrução ficará em um único local. No executável, portanto, a função **provavelmente não** existirá.
- Já se a função for **grande** (contendo muitas instruções) e estiver sendo chamada em **diversos lugares**, então o compilador irá **avaliar** até que ponto a operação de troca aumentará o tamanho do executável. Desse modo a função poderá **existir ou não** no executável.

O critério de avaliação do compilador portanto é baseado em um critério de "custo/benefício", onde são levados em conta os ganhos de performance, mas também o tamanho final do executável. Mas **cada compilador** pode ter seu **próprio critério** de avaliação. Assim podemos usar funções **inline** para tornar mais legível e simples o uso de determinadas operações, **nomeando-as**.

*Vejam alguns exemplos:*

// Determinando o menor de dois números inteiros:

```
inline bool Minimo( int a, int b)
{
    return ( a < b ) ? a : b;
}
```

✂ Uma operação, como a do exemplo acima, não é **tão rapidamente legível quando o seu nome**. Quando chamamos "Minimo" o próprio nome diz o que vai ser feito. Ao passo que na implementação (a comparação) precisamos olhar o que está sendo feito.

Se, ao invés da função, inseríssemos essa operação em todos os lugares onde fosse necessária, teríamos sempre que fazer um comentário para que o seu sentido ficasse claro.

Muito melhor uma função **inline**:

além da *clareza*, pelo fato de a operação ter recebido um **nome** (que já indica a sua **finalidade**), a função mantém esse código em um **único** lugar, facilitando manutenções futuras.

E, também, as **explicações (comentários)** sobre essa operação estarão em um **único** lugar (a função).

Podemos também usar funções **inline** para garantir que determinadas operações sejam sempre realizadas de modo completo, ganhando assim em segurança.

- Por exemplo: ao usar a função **strncpy**, corremos um risco. Se a origem da cópia for maior que a quantidade de bytes passada no terceiro parâmetro (tamanho), essa função irá copiar até o último byte. Não ultrapassará o tamanho de memória indicado, mas também não colocará o terminador zero no último byte.

Portanto, a seguinte função **inline** seria útil:

```
inline void copiaString ( char * Destino, const char * Origem , int bytes )
{
    strncpy ( Destino , Origem , bytes );
    Destino [ bytes ] = '\0'; // garante o terminador zero no último byte
}
```

Além disso podemos querer **isolar e nomear** determinados trechos internos de uma função maior apenas para dar legibilidade à função maior.

- A escrita de funções tem a vantagem de melhorar a legibilidade ao dar um **nome** a trechos de código maiores ou mais complexos e também por isolar o **começo e fim** de cada um.
- Contudo em **C** isso era mais oneroso, pois perdemos eficiência efetuando chamadas de função para um código que na realidade só será executado naquele ponto.
- Em **C++** podemos criar **funções inline**. E isso **pode** resolver o problema da perda de *performance* ocasionado pelo **custo de chamada** de função.

**Exemplificando...**

```
// ...
int a, b;
// .....
int c = Minimo ( a , b ); // ➔ esta linha seria substituída por:
    int c = a < b ? a : b;
int c = Minimo ( ++a , --b ); // ➔ seria substituído por:
++a ;           // (1)
--b;           // (2)
```



Notar que primeiro são resolvidas as **pendências (1) e (2)**, para depois ser realizada a operação prevista pela **inline** "Mínimo":

```
int c = a < b ? a : b ; // (3)
```

```
.....
```

### 5.6.9 • Funções recursivas.

Tanto C como C++ admitem que uma função chame a si mesma. Essa característica é denominada **recursividade**.

Na prática, quando uma função chama a si mesma, temos uma situação **semelhante** a um **laço**, já que o conjunto de instruções contido pela função será repetido até que uma **condição de fim** ocorra.

Nos casos **mais simples** a semelhança entre uma função recursiva e um laço é muito grande.

Já em outras situações, mais complexas, embora o princípio básico seja o mesmo (repetir um conjunto de instruções até que uma condição se cumpra), o uso de funções recursivas, ao invés de laços, simplificará muito a lógica necessária.

Vejamos inicialmente um exemplo simples, onde o uso de um **laço for** ou de uma **função recursiva** quase se equivalem.

Assim, para melhor comparar os dois métodos, vamos escrever duas funções de cálculo de fatorial.

A **primeira** usará o **laço for**; e a **segunda** será uma **função recursiva**.

```
#include <iostream>
```

```
// 1) esta versão do cálculo de Fatorial usa o laço for:
```

```
unsigned long long Fatorial_Usando_For( unsigned int Numero )
{
    unsigned long long Fatorial = 1;
    for ( ; Numero > 1 ; Numero-- )
        Fatorial *= Numero;
    return Fatorial;
}
```

```
// 2) esta versão do cálculo de Fatorial usa uma função recursiva:
```

```
unsigned long long Fatorial_Recursiva ( unsigned int Numero )
{
    if ( Numero > 1 )
        return Numero * Fatorial_Recursiva( Numero - 1 );
    else
        return 1;
}
```

Que também pode ser escrita assim:

```
unsigned long long Fatorial_Recursiva ( unsigned int Numero )
{
    return ( numero > 1 ) ? Numero * Fatorial_Recursiva( Numero - 1 ) : 1;
}
```

```
/*
```

A função acima chamará a si mesma até que Numero fique igual a 1.

Só então começará a retornar. Então, se o número for 3, ocorrerá que:

```

- ela chamará a si mesma passando 3 menos 1, como parâmetro;
- agora, ela receberá 2 como parâmetro e novamente chamará a si mesma, desta vez
  passando 2 menos 1 como parâmetro;
- agora, ela receberá 1 como parâmetro e teremos o primeiro retorno: 1.
- O retorno (1), será multiplicado pelo retorno anterior (2), retornando 2;
- O resultado anterior (2) será multiplicado por 3, e temos o retorno final: 6
*/

int main()
{
    cout << int( Fatorial_Usando_For ( 6 ) ) << endl;
    cout << int( Fatorial_Recursiva ( 6 ) ) << endl;
    cout << int( Fatorial_Usando_For ( 10 ) ) << endl;
    cout << int( Fatorial_Recursiva ( 10 ) ) << endl;
    return 0;
}

/* RESULTADO:

                                720
                                720
                                3628800
                                3628800

*/

```

Assim, os resultados são os mesmos para a versão **recursiva** e para a versão que usa o laço **for**.

Vejam agora a **diferença** entre um laço e uma função recursiva.

Uma função pode ter variáveis automáticas (parâmetros, variáveis internas e área de retorno).



Assim a cada nova chamada é criada uma **nova instância** desse conjunto de variáveis.



E poderá ocorrer um estouro de pilha ("**stack overflow**") caso a função tenha muitas variáveis e/ou a **condição de fim** da recursividade esteja muito distante.

Pois um número muito grande de chamadas continuará **pendente**, e as variáveis correspondentes à **cada chamada** continuarão **reservadas**, aumentando continuamente a ocupação da pilha.

Por isso é preciso um cuidado maior quando usamos funções recursivas.

Mas há situações em que sua utilização **simplificará** enormemente o algoritmo da solução.


São casos em que laços convencionais levariam a um grande aumento de testes de decisão e desvios sucessivos.

Um exemplo clássico são os algoritmos de classificação de dados.

Em teoria, seria possível escrevê-los sem usar funções recursivas (usando laços). Mas isso aumentaria o trabalho e implicaria em sequências lógicas muito complicadas.

Por outro, em algoritmos desse tipo, a recursividade é limitada por sua própria natureza, não existindo o menor perigo de estouro de pilha.

### 5.6.10 • Funções sobrecarregadas

 Se escrevermos diversas funções usando um mesmo nome, podemos dizer que temos funções **sobrecarregada**;  
Isto é: a função tem mais do que uma forma (ou várias versões) para ser chamada.

#### E como escrever funções diferentes usando um mesmo nome?

A regra básica da sobrecarga é que exista alguma diferença quanto à **lista de parâmetros**, pois do contrário o compilador não teria como distinguir qual versão está sendo chamada.

*Exemplo:*

```
versão 1 - int SomaNumeros ( int A , int B ) ;
versão 2 - float SomaNumeros (float A, float B) ;
.....
int main( )
{
    int A = SomaNumeros ( 5, 4 ); /* será chamada a VERSÃO 1, pois é
                                   esta que recebe dois parâmetros do tipo int */

    float B = SomaNumeros ( 5.3F, 4.4F ); /* será chamada a VERSÃO 2,
                                             que recebe dois parâmetros do tipo float */

    return 0;
}
```

## 5.7 • Declarações constexpr (em C++11 e em C++14)

Uma declaração **constexpr** visa pedir ao compilador que ele **tente** resolver uma expressão durante a própria **compilação (e não em tempo de execução)**. Mas, se isso **não** for possível, a expressão só será resolvida em tempo de **execução**.

Declarações **constexpr** podem ser aplicadas tanto a **funções** como a **objetos**.


### 5.7.1 • Funções constexpr


#### 5.7.1.1 • Em C++11

 Em **C++11**, uma **função constexpr** só pode ter **uma única** linha de código.

*Exemplo:*

```
constexpr unsigned long long fatorial_cpp11 (unsigned int n)
{
    return n > 1 ? n * fatorial_cpp11( n - 1 ) : 1;
}
```

 A função "fatorial\_cpp11" poderá ser resolvida em tempo de **compilação** se receber como argumento um valor **constante** para alimentar o parâmetro "n". Mas só poderá ser resolvida em tempo de **execução** se esse argumento **não** for constante.


 Além disso, em **C++11**, funções **constexpr** **não** podem ser **void**.


*Exemplo:*

```
#include <iostream>
int main()
{
    // → abaixo, como 5 é uma constante, provavelmente "fatorial_cpp11"
    //      será resolvida em tempo de compilação:
    unsigned long long fat1 = fatorial_cpp11(5);
    std::cout << fat1 << '\n';

    // → mas, abaixo, isso não será possível:
    unsigned int x;
    std::cin >> x;
    // "x" só terá o seu valor conhecido em tempo de execução.
    // Logo, "fatorial_cpp11" só poderá ser resolvida em tempo de execução:
    unsigned long long fat2 = fatorial_cpp11(x);
    std::cout << fat2 << '\n';
    return 0;
}
```

### 5.7.1.2 • Em C++14


 A partir de **C++14**, temos uma flexibilização - e uma função **constexpr** passa a poder ter **mais do que uma** linha de instrução.


 E, também, ao contrário de **C++11**, em **C++14** funções *constexpr* **podem** ser **void**.

#### Exemplo:

```
constexpr unsigned long long fatorial_cpp14 (unsigned int n)
{
    unsigned long long result = 1;
    for ( ; n>1 ; --n)
        result *= n;
    return result;
}
```


Quanto ao uso, aplica-se o mesmo que foi dito para **C++11**: se o parâmetro **n** for preenchido com um argumento constante, "fatorial\_cpp14" poderá ser resolvida em tempo de compilação; do contrário só em tempo de execução

 Aqui, pode-se usar o mesmo exemplo acima, trocando "fatorial\_cpp11" por "fatorial\_cpp14" - e a avaliação do compilador será a mesma.

 **Anote:** se for possível resolver uma função em tempo de **compilação**, teremos um ganho de **performance** que não devemos desprezar (afinal, a função pode estar sendo chamada **inúmeras** vezes dentro de um **laço**).

### 5.7.2 • Objetos constexpr e diferença para *const*

Um objeto declarado como **const** e **também** um objeto declarado como **constexpr** obrigatoriamente devem ser **inicializados**, e **não** poderão ser **alterados** posteriormente através de atribuição. **Nisso**, *const* e *constexpr* se assemelham.

 **Contudo**, a **inicialização** de um objeto declarado como **const** pode ser **diferente** da **inicialização** de um objeto declarado como **constexpr**. *A saber:*

- Um objeto **const** pode ser inicializado com uma expressão constante (tempo de compilação), mas **também** pode ser inicializado com uma expressão que só terá seu valor conhecido em tempo de **execução**.
- Já um objeto declarado como **constexpr** só poderá ser inicializado com um valor **constante**, conhecido em tempo de **compilação**.

### Exemplo:


```
#include <iostream>
#include <cmath> // funções matemáticas...

// cria sinônimos simplificados:
using ulonglong = unsigned long long;
using uint = unsigned int;
constexpr ulonglong fatorial_cpp14 (uint n)
{
    ulonglong result = 1;
    for ( ; n>1 ; --n)
        result *= n;
    return result;
}

int main()
{
    constexpr int x1 = 5; // OK! constexpr inicializado com a constante 5
    const int x2 = 5; // OK! const inicializado com um valor constante.
    // x1 = 10; // Erro! constexpr não pode ser alterada.
    // x2 = 10; // Erro! const também não pode ser alterada
    // → mas, abaixo, temos a diferença entre os dois modos de inicialização:
    // constexpr double d1 = std::pow(10,2); // ERRO! A função std::pow (calcula potência)
    // não é constexpr. Logo, apesar de, nesse exemplo, receber argumentos
    // constantes, ela só será resolvida em tempo de execução.
    const double d2 = std::pow(10,2) // OK! const pode ser inicializada na execução.
    // nas duas declarações abaixo não temos problemas:
    constexpr ulonglong fat1 = fatorial_cpp14(5); // OK! "constexpr fat1"
    // inicializada com uma função constexpr que recebe argumento constante
    const ulonglong fat2 = fatorial_cpp14(5) // OK! também inicializada em compilação
    // → mas, abaixo, temos novamente diferença entre os dois modos de inicialização:
    uint x = 10;
    // ...
    constexpr ulonglong fat3 = fatorial_cpp14(x); // ERRO! "x" não é uma constante
    const ulonglong fat4 = fatorial_cpp14(x) // OK! "fat4" é inicializada em execução

    return 0;
}
```

## 5.8 • Lambdas (C++11)

 Lambdas são **funções sem nome** que você pode escrever **inline** em seu código fonte e podem ser utilizadas em locais onde funções tradicionais seriam usadas.

As sintaxes para a criação das *lambdas* são as seguintes:

```
[captura](parametros) -> retorno {corpo}
[captura](parametros) {corpo} // o tipo de retorno é deduzido do return de {corpo}
[captura]{corpo} // parâmetros não são necessários
```



Onde:

- **[captura]** – indica quais variáveis locais serão "capturadas" para uso no **corpo** (a parte funcional) da expressão lambda. Pode assumir as seguintes formas:
  - **[ ]** : colchetes vazios; nada é capturado.
  - **[ = ]** : todas as variáveis locais são capturadas por cópia de valor.
  - **[ & ]** : todas as variáveis locais são capturadas por **referência**; isso significa que se uma variável capturada dessa forma for alterada no **corpo** da expressão lambda (a **função**), ela estará sendo **alterada** no escopo de quem invocou a expressão lambda. Veremos referências com detalhes no capítulo 6, página 186 ("*Ponteiros e Referências*").
  - **[ variavel\_1, ..., variavel\_n ]** : captura **apenas** as variáveis constantes na lista de captura (uma ou mais).
  - variáveis *static* e *extern*(globais) são naturalmente (**implicitamente**) capturadas. Basta que estejam visíveis no escopo em que a expressão é invocada.
- **(parâmetros)** - parâmetros para a parte funcional da expressão.
- **-> (retorno)** – indica formalmente o tipo do retorno da parte funcional.
- **{corpo}** – é a parte funcional da expressão, isto é, aquilo que será executado.


**Exemplo:**

```
#include <iostream>
int main()
{
    int a = 1, b=2;
    // captura a e b por valor; retorno da lambda deduzido do return do corpo (int):
    int x1 = [=]() { return a + b; } ();
    // captura a e b por valor; retorno especificado explicitamente como double:
    double x2 = [=]() -> double { return double(a + b)/2; } ();
    // não captura nada; "a" e "b" são copiados para os parâmetros "x" e "y":
    int x3 = [](int x, int y) { return x + y; } (a, b);

    std::cout << "int x1 = " << x1 << ", double x2 = " << x2 << ", int x3 = " << x3 << '\n'
    // Resultado: int x1 = 3, double x2 = 1.5, int x3 = 3
    return 0;
}
```

Veremos *mais exemplos* abaixo.


### 5.8.1 • Expressão lambda através de uma variável auto

 Uma expressão lambda embora não possua um nome pode ser chamada através de uma **variável auto**. **Exemplo:**

```
#include <iostream>
double menor(double a, double b)
{
    return a < b ? a : b;
}
// "lambdaMenor" armazena uma expressão lambda
auto lambdaMenor = [](double a, double b) -> double {return a < b ? a : b;};

int main()
{
    std::cout << menor(10, 20) << "\n\n"; // chama diretamente a função "menor"
    std::cout << lambdaMenor(10, 20) << '\n'; // chama a expressão lambda através
    // da variável "lambdaMenor"; com isso a função "menor" será chamada pela lambda
    return 0;
}
```

## 5.8.2 • Capturando variáveis em lambdas

 **OBS:** variáveis **static** e **globais** são capturadas automaticamente (**implicitamente**), desde que estejam visíveis no escopo em que a expressão **lambda** é invocada.

```
#include <iostream>

int main()
{
    int a = 1, b = 2;
    static int c = 3;


    std::cout << "Captura todas as váriaveis por cópia:\n";
    auto fun1 = [=](int n) { std::cout << n * a * b << '\n'; };
    fun1(10); // imprime 20 (10 * 1 * 2)


    std::cout << "\n\nCaptura todas as variaveis por referência:\n";
    auto fun2 = [&](int n) { std::cout << n * a * b << '\n'; };
    fun2(10); // imprime 20 (10 * 1 * 2)

    std::cout << "\n\nQuando a variavel é static ou global não precisa capturar,\n";
    std::cout << "pois são capturadas por referência automaticamente.\n";
    std::cout << "No exemplo abaixo, 'a' e 'c' ('c' implicitamente) são capturadas: \n";

    auto fun3 = [&a](int n)
    {
        std::cout << n * a * c << '\n';
    };
    fun3(10); // imprime 30 (10 * 1 * 3)
    return 0;
}
```

## 5.8.3 • Generic lambdas (C++ 14)

 No padrão **C++11**, parâmetros de funções lambda precisavam ser declarados com tipos **explícitos**.

 Mas no padrão **C++14** é permitido que os parâmetros e retorno de funções lambda sejam declarados como **auto**. **Exemplo:**

```
#include <iostream>

auto soma = [](auto a, auto b) -> auto { return a + b; };

int main()
{
    std::string nome = "Agit ", sobrenome = "Informatica";
    double x = 20.5, y = 79.5;

    // Qual será o tipo dos parâmetros "a" e "b" e do retorno na expressão lambda,
    // nas chamadas abaixo?

    std::cout << soma(nome, sobrenome) << '\n'; // "a", "b" e o retorno são do tipo string
    std::cout << soma(x, y) << '\n'; // "a", "b" e o retorno são do tipo double.

    return 0;
}
```

## 5.9 • Tomadas de decisão (detalhamento)

### 5.9.1 • if / else

Já vimos, em exemplos acima, avaliações de condição para tomadas de decisão através de { if (...) ... [ else ... ] }.

Em sua **forma geral**, esse controle de fluxo é definido como:

```
{ if ( <condição> ) ... [ else ... ] }
```

mas o else pode ser acompanhado de um novo if:

```
{ if ( <condição> ) ... [ else if ( <condição> ) ... ] [ else ... ] }
```

**Exemplo:**

```
if ( valor_pago < valor_venda )
{
    // .....
}
else if ( valor_pago > valor_venda )
{
    // .....
}
else // ... iguais
{
    // .....
}
```

**Algumas questões básicas a ressaltar sobre o if:**

- A **condição** a avaliar deve estar entre **parênteses**.
- Não necessariamente o "**else**" precisa estar presente.
- Se existir um "**else**" (inclusive na forma combinada "**else if**") ele não pode aparecer isoladamente, devendo sempre ser **precedido** pelo **if**.
- Existindo mais do que uma instrução a ser executada caso uma condição seja verdadeira ("if" ou "else if") ou falsa ("else"), essas instruções deverão estar entre **chaves** (como nos **dois exemplos acima**).
  - As **chaves** delimitam assim um **bloco de instruções** que é executado em determinada condição.
- Caso exista **apenas uma** instrução, as chaves podem ser **omitidas**, como foi feito mais acima no exemplo da função "*Mínimo*".
 

```
if ( a < b ) return a ;
else return b ;
```

#### 5.9.1.1 • Cuidados a tomar com o if

- **Nenhuma instrução associada.** No caso do **if** isto não faz sentido.
- Assim, o código abaixo provocará uma **advertência** do compilador (algo como "*é isso mesmo que você quer fazer?*"):

```
if ( condicao ) ; // o ponto e vírgula indica que
                // nenhuma instrução foi associada ao if
```



**Ao contrário de um laço, um if executará uma única avaliação da condição.**

Assim sendo, *tanto faz que essa avaliação retorne falso ou verdadeiro: será executado aquilo que estiver **após** a linha do if em ambos os casos.*

- Em um **laço** podemos ter::

```
while ( Pausa ( ) ) ;
```

- E **isto pode fazer sentido**, porque a função **Pausa** (que deverá retornar verdadeiro ou falso), poderá ser executada  **muitas vezes**  (até que ela retorne falso, provocando o encerramento desse laço).

- Já se fizéssemos :

```
if ( Pausa( ) ) ;
```

- Seria **a mesma** coisa que, simplesmente:

```
Pausa( ) ;
```

- Pois a função, em **ambos** os casos, seria executada uma **única vez**, sem que o seu **retorno** tenha **qualquer consequência**.
- Assim, uma linha de instrução como essa [ "if ( Pausa( ) ) ; " ], ou é o resultado de uma distração (e por isso o compilador emite um aviso) ou não passa de uma grande bobagem.

### Precauções adicionais:

#### 1) Aninhamento.

- Ao usar um **if** dentro de outro, é aconselhável utilizar blocos seja para tornar mais rápida a releitura do código, seja para evitar enganos. **Assim:**

```
int x , y ;
.....
if ( x > 5 )
    y = x ; // Esta é uma instrução completa terminada no ponto e vírgula:
.....          // "se 'x' for maior que 5, copie o valor de 'x' para 'y' "
```

// Na linha anterior não tivemos novidades.

// Vamos agora inserir um if aninhado

```
if ( x > 5 )
    if ( y == 0 )
        y = x ; // Aqui termina uma instrução completa:
                // "se 'x' for maior que 5,
                // e se 'y' for igual a zero, copie o valor de 'x' para 'y' "
```

// E agora vamos introduzir um else:

```
if ( x > 5 )
    if ( y == 0 )
        y = x ; // Aqui termina uma instrução completa:
                // "se 'x' for maior que 5,
                // e se 'y' for igual a zero, copie o valor de 'x' para 'y' "
    else
        x = y ; // Aqui termina outra instrução completa:
                // "se 'x' for maior que 5,
                // e se 'y' não for igual a zero, copie o valor de 'y' para 'x' "
```

- O trecho acima está **correto**. Mas poderia ser escrito **também** assim:

```
if ( x > 5 ) // Se 'x' for maior que 5 execute o bloco abaixo.
{
    if ( y == 0 ) // Se 'y' for igual a zero execute a instrução abaixo.
        y = x ;
    else // Do contrário ( 'y' diferente de zero), execute a instrução abaixo.
```

```

        x = y ;
    }

```

- A **segunda** alternativa é **melhor** porque é mais clara e também porque ajuda a evitar distrações. E o que ela deixa claro é que **não** queríamos fazer **isto**:

// Terceira alternativa (que não é a desejada neste caso):

```
if ( x > 5 )    // Se 'x' for maior que 5 execute o bloco abaixo.
```

```
{
```

```
    if ( y == 0 )    // se 'y' for diferente de zero, não haverá nada a fazer.
        y = x ;
```

```
}
```

```
else    // Do contrário, ( 'x' menor ou igual a 5) execute a instrução abaixo:
```

```
    x = y ;
```

- Um **else** está sempre relacionado ao último **if** que foi empregado. Por isso as duas primeiras alternativas levam ao mesmo resultado. Mas perceba que quando **não** usamos as chaves abrimos mão de ser **taxativos** (e, caso o programa apresente problemas, alguém que releia o código poderá ter dúvidas e perder tempo analisando se não deveria ter sido usada a terceira alternativa – embora o problema não esteja nesse ponto).
- Com as chaves estamos optando de forma **intencional e direta** por uma determinada alternativa lógica. Estamos afirmando: “*não deve haver qualquer dúvida*”.

## 2) Utilização do operador de atribuição dentro da condição.

O fato de o operador de atribuição utilizar como símbolo um sinal de igual e o operador de comparação por igualdade utilizar dois sinais de igual seguidos, costuma ser uma fonte de erros que ocorrem com alguma frequência entre programadores iniciantes na linguagem.

*Por exemplo:*

```
if ( x == 1 )    // aqui foi usado o operador de igualdade
    .....    // (simbolizado por um dois sinais de igual)
```

**É muito diferente de:**

```
if ( x = 1 )    // e aqui foi usado o operador de atribuição
    .....    // (simbolizado por um único sinal de igual)
```

**E o que, exatamente, será provocado pela linha acima?**

- Em **primeiro** lugar, o número inteiro **1** será **atribuído** à variável **'x'**.
- Em **segundo** lugar, será executado o seguinte **teste de condição**:
  - Pergunta: o resultado da operação anterior é **verdadeiro** (diferente de zero) ou **falso** (igual a zero) ?
  - Como o resultado da operação de atribuição é **1** (verdadeiro) então o resultado lógico é “a condição é verdadeira”.  
Portanto, serão executadas as instruções associadas ao **“if”**.
  - E, como a atribuição é feita com um valor constante (**1**), o **resultado** também será **constante**. Desse modo, o **“if”** não faz o menor sentido: pois simplesmente as instruções a ele associadas serão **sempre** executadas.



E a única conclusão possível, nesse caso, é que ocorreu um descuido do programador, esquecendo de inserir o segundo sinal de igual para que fosse estabelecida a **comparação: if ( x == 1 )**.

A linguagem **permite** esse modo de escrita porque em alguns casos isso pode servir para economizar linhas de código. Mas, obviamente, isso só fará **sentido** se o **resultado da operação não for constante**.

*Vejamos o seguinte exemplo:*

```
int x , y ;
.....
if ( x = y )
.....
```

No caso acima, ocorrerá o seguinte:

- Em primeiro lugar, o valor de '**y**' será copiado para '**x**' (atribuição).
- Agora, o resultado será testado (verdadeiro ou falso?).
- Como '**y**' é uma **variável**, em alguns casos poderá conter valores diferentes de zero (levando à decisão: verdadeiro); e, em outros, poderá estar valendo zero (levando à decisão: falso).
- **Isto** faz sentido.

Esse modo de escrita permitiu economizar uma linha adicional de código fonte. Senão, teríamos que escrever as duas operações (atribuição e comparação com zero) como linhas de instrução distintas:

```
x = y ; // Copie o valor de 'y' para 'x'.
if ( x ) // Agora, analise se 'x' é verdadeiro (diferente de zero)
.....
```

Do ponto de vista do resultado final não há diferença entre uma forma e outra. Mas é verdade também que, quando usamos o modo econômico, fica a **dúvida**: foi **intencional** ou também aqui tivemos um **descuido** ?

De todo modo, se utilizada uma constante, temos **certeza**: foi um **descuido**.

Mas, se utilizada uma variável, pode ser intencional ou não.

E, se o programa estiver apresentando problemas, é muito possível que alguém suspeite dessa linha de código e perca tempo analisando a situação lógica.

Por isso mesmo, **alguns compiladores** emitem um **aviso**, dizendo mais ou menos o seguinte: "*esse uso é intencional?*".

E, para não receber essa mensagem de advertência, precisaremos escrever do seguinte modo:

```
if ( ( x = y ) )
.....
```

Escrevendo dessa forma, estamos dizendo ao compilador (e a outros programadores que releiam esse código):

*"Eu sei o que estou fazendo: propositalmente, esta linha contém **duas** operações, primeiro uma **atribuição** (parênteses **interno**) e depois um **teste de condição** (parênteses **externo**)."*

Infelizmente, nem todos os compiladores emitem esse aviso (já que, pelo padrão da linguagem, eles não são obrigados a fazer isso).



Sempre prefira a **clareza**. Faça com que suas intenções estejam presentes no código: ou não utilize operações de atribuição dentro de testes de condição, ou então seja o mais explícito possível, utilizando os parênteses extras.

### 5.9.1.2 • A precaução com a atribuição também se aplica aos laços.



Tome um cuidado **especial** ao utilizar esse modo de escrita em **laços**. Pois, além de atribuições constantes serem um erro (do mesmo modo que no "if"), precisaremos prestar mais atenção mesmo nas atribuições **variáveis**.

*Por exemplo: a linha abaixo faz sentido ?*

```
while ( ( x = y ) ) // copia "y" para "x" e depois avalia o resultado: verdadeiro ou falso?
.....
```

*Depende:*

- 1) Se o valor de 'y' for **alterado** em uma das instruções associadas ao **laço** e **supondo-se** que em algum momento 'y' fique **falso** (para que não tenhamos aí um laço infinito), então faz sentido.
- 2) Do contrário isso não faz sentido. Pois sempre chegaremos ao laço com 'y' verdadeiro ou falso. E então, já que o seu valor não será alterado, temos apenas duas possibilidades:
  - Nas ocasiões em que estiver verdadeiro o laço será **infinito**.
  - E, quando estiver falso, o laço simplesmente **não** será executado **sequer uma vez**.

### 5.9.2 • switch

Para melhor entender em que situações podemos usar este controlador de fluxo, vamos ver primeiro um exemplo com [ **if ... else ...** ]:

```
int x ;
// ...
if ( x == 1 )
{
    // ...
}
else if ( x == 2 )
{
    // ...
}
else
{
    // ...
}
```

O conjunto [ **if ... else if ... else ...** ] acima, tem as seguintes características:

- a. O valor que está sendo analisado é sempre **o mesmo**. No exemplo acima, esse valor é o resultado da leitura da variável "**x**".
- b. Poderia ser o valor resultante de qualquer expressão, desde que seja um valor inteiro. Como já sabemos, nessa condição temos: int, char, short, long e enum.
- c. O operador empregado é sempre o operador relacional de **igualdade**, **não** sendo usado **qualquer outro** operador relacional ou lógico.
- d. E apenas **uma** comparação por igualdade é feita em cada uso do **if**.
- e. O valor analisado é sempre comparado a um **valor constante** (1 e 2, no exemplo).

Nestes casos, podemos usar:

```

switch ( x )
{
    case 1:
        // ...
        break;
    case 2:
        // ...
        break;
    default:
        // ...
}
// próxima instrução após o switch

```

- Nesse exemplo, como o único valor que é avaliado é o valor de '**x**', o avaliado é colocado em um **único lugar: switch ( x )**.
- Em seguida, o valor de '**x**' é testado para as constantes **1 [ case 1: ]** e **2 [ case 2: ]**.
  - Se o valor de '**x**' bater com um desses valores, será executado o código associado ao respectivo rótulo **case**.
  - Do contrário, é executado o código associado ao rótulo **default**.
- O **switch** serve para situações simples como essa. Ele é restrito, pois não tem pretensões de ser um **if...**
- Por exemplo, se um determinado rótulo **case** satisfizer a condição e, no código associado, não houver um desvio **break** (ou um **return**), a execução continuará no código associado ao próximo **case** ou ao **default**.
  - Assim, se, no exemplo acima, o valor de '**x**' fosse **1** e não houvesse o desvio **break**, seria executado **também** o código associado ao rótulo **case 2**. Do mesmo modo, se também aqui não houvesse um desvio, seria executado o código associado ao rótulo **default**.

**Considerando este exemplo:**

```

int x = 1 ;
switch ( x )
{
    case 1:
        std::cout << "1 - " ;
    case 2:
        std::cout << "2 - " ;
    default:
        std::cout << "menor que 1 ou maior que 2" ;
}

```

**Seria impresso:**

1 - 2 - menor que 1 ou maior que 2

## 5.10 • Laços.

### 5.10.1 • Porque usar laços estruturados

Já vimos os laços **while** (seção **3.3.2.1**, página **82**, acima) e **for** (seção **3.3.2.3**, página **86**, acima).

Temos aí dois dos **laços estruturados** oferecidos por **C** e **C++**. Quando dizemos "estruturados", isso significa que o seu fluxo não é estabelecido por desvios executados



manualmente pelo programador, e sim por uma **estrutura de controle de laço**, com uma sintaxe determinada.



#### Aviso aos programadores *assembler*.

Um programador *assembler*, ao ver construções como o **while** ou o **for** pela primeira vez, provavelmente irá considerá-las estranhas. Isso porque, nas linguagens de máquina, não existem construções assim. Existem **apenas saltos**.

As linguagens **C** e **C++** também permitem que problemas desse tipo sejam resolvidos por **saltos** (ou desvios livres).

É o que veremos no **exemplo abaixo**, onde criamos um laço para o cálculo de fatorial através de um desvio direto (**goto**).

Um laço estruturado, ao ser traduzido para linguagem de máquina, será resolvido (como não poderia deixar de ser) através de saltos, de modo que, no fim das contas, ficará muito parecido com o exemplo abaixo.

Contudo, considere que, **no código fonte**, os **laços estruturados**:

- **reduzem** a quantidade de **erros lógicos** potenciais, pois a probabilidade de ocorrência desse tipo de erro é maior quando **saltamos** livremente para **qualquer parte do código**;
- melhoram enormemente a **legibilidade** do código.

#### Exemplos com laços estruturados e laços baseados em saltos.

a. Relembrando o exemplo de **laço** usando o **for** para cálculo de **fatorial**:

```
int num = 10 , result ;
for ( result = 1 ; num > 1 ; --num )
    result *= num ; // instrução a executar se condição verdadeira
```

b. Exemplo de **laço** usando o **goto** (desvio livre) para cálculo de **fatorial**:

```
int num = 10 , result = 1 ; // início
➔ AVALIA : // "AVALIA" é um rótulo que referencia a próxima instrução
// permitindo que um salto desvie o fluxo para este ponto:

if ( num > 1 ) // condição de continuidade
{
    result *= num ; // processa...
    --num ; // progressão
    goto AVALIA ; // volta para a avaliação da condição, permitindo
// a repetição das instruções, em um laço.
}
// quando a condição (num>1) for falsa, o fluxo seguirá abaixo.
```



**Laços estruturados são melhores do que laços estabelecidos através de saltos. Reduzem erros lógicos e tornam o código bem mais legível.**

### 5.10.2 • Laço do ... while

Além dos laços **while** e **for**, temos ainda o laço [ **do ... while ( <condição> );** ].

Ele é parecido com o **while**, mas tem uma grande diferença:

<pre>int x ; // ... while ( x &gt; 1 ) {     // &lt;instruções&gt; ... }</pre>	<pre>do {     // &lt;instruções&gt; ... } while ( x &gt; 1 ) ;</pre>
<p>No <b>while</b> temos uma condição <b>a priori</b>: isto é, ela é avaliada no <b>início</b> do laço, e, por isso, se estiver <b>inicialmente falsa</b> as &lt;instruções&gt; não serão executadas.</p>	<p><b>do ... while</b> : condição <b>a posteriori</b>: isto é, a condição só é avaliada ao <b>final</b> do laço e com isso as &lt;instruções&gt; serão executadas <b>pelo menos uma vez</b>.</p>

### 5.10.3 • Desvios por salto incondicional.

#### 5.10.3.1 • Desvio *return*.

Um **return** provoca um retorno **de função**. Isto significa que ele retorna sempre ao ponto imediatamente seguinte à chamada de função. Então o **return** é um desvio **estruturado**, pois o programador não é livre para indicar o alvo desse desvio.

Se a função retornar valor, o **return** copiará esse valor para a memória de retorno, antes de proceder ao desvio para retornar.

Um **return** pode ser incluído em qualquer ponto de uma função e não tem, em si mesmo, nenhum recurso para teste de condição.

Contudo pode ser associado a algum outro controle de fluxo que permita um teste de condição, como, por exemplo o **if**:

```
if ( Condicao )
    return ;
```

#### 5.10.3.2 • Desvio *break*.

Um **break** pode **interromper** um laço ( “for”, “while” e “do ... while”) ou o controle de decisão “switch”.

Em um **laço**, o **break** provoca um salto para a **próxima instrução após o fim** do laço (exatamente como ocorreria se o laço tivesse sido encerrado normalmente).

Assim sendo, o **break** também é um desvio **estruturado**, pois o programador não é livre para indicar o alvo do salto.

O **break** também não tem um recurso próprio para teste de condição. Mas pode ser associado a qualquer outro controle de fluxo que ofereça o teste de condição.

#### *Exemplo:*

- No exemplo abaixo, pede-se ao usuário que informe as notas de um aluno para cálculo de média.
- Um laço é usado para que o usuário informe a próxima nota até atingir o máximo de notas.
- Se, contudo, o aluno tiver menos notas do que o máximo, o operador poderá interromper o laço a qualquer momento digitando um número negativo. Para esta interrupção será usado um **break**.

```
#include <iostream>
constexpr int MAX_NOTAS = 6;
int main( )
{
    int TotalNotas = 0;
```

```

double Soma = 0;
double NotaDaVez = 0;
std::cout << "Calculo de Media do Aluno\n" ;
std::cout << "Digite ate " << MAX_NOTAS <<
    " notas, entre zero e dez\n" ;
std::cout << "(para interromper, digite -1)" << "\n\n";
while ( TotalNotas < MAX_NOTAS )
{
    std::cin >> NotaDaVez;

    if ( NotaDaVez < 0 ) // Número negativo: não há mais notas a informar.
        break ; // Salta para a primeira instrução após o fim do laço.

    TotalNotas++ ;
    Soma += NotaDaVez ;
}

std::cout << '\n' ;
cout << "Total de Notas Informadas: " << TotalNotas << "\n\n";
if ( TotalNotas > 0 ) // Prevenir divisão por zero.
{
    cout.width(5) ; // Largura máxima do número a imprimir (incluindo decimais e ponto)
    cout.precision(3) ; // Quantidade de casas decimais (mais o ponto) a imprimir
    // Neste caso, será arredondado para duas casas decimais
    cout << "Media = " << Soma / TotalNotas << endl;
}
else
    cout << "Media zero por ausencia de notas" << endl ;
return 0;
}

/* RESULTADO (notas informadas: 7, 6 e 9)
    Calculo de Media do Aluno
    Digite ate 6 notas, entre zero e dez
    (para interromper, digite -1)
    7
    6
    9
    -1
    Total de Notas Informadas: 3
    Media = 7.33
*/

```

### 5.10.3.3 • Desvio *continue*.

Conceitualmente, um **continue** provoca um salto para o **fim de um laço**. Mas, na implementação, os compiladores podem levar em conta as diferenças entre os diversos laços para, assim, gerarem um código de máquina mais eficiente.

Em um laço “**for**” este **salto** tem como alvo a linha onde está o segmento de **progressão** do laço. Em, seguida será executada a **condição de continuidade**.

Em um laço “**while**”, saltar para o fim significa **voltar** para nova execução da **condição de continuidade**.

**Resumindo:**

- em um laço “**for**”, um **continue** despreza (saltando) todas as linhas entre ele e a **progressão**; então a progressão é executada imediatamente e, em seguida, ocorre o salto normal para o teste de **condição** no início do laço.
- em um laço “**while**”, um **continue** despreza (saltando) todas as linhas entre ele e o **fim do laço**: então deve ser executado imediatamente o **teste de condição** no início do laço (não faria sentido saltar para o fim para então usar o salto normal para o início).
- em um laço “**do ... while**” o salto para o fim coincide com o **teste de condição**.

O **continue** também é um desvio **estruturado**, pois o programador não é livre para indicar o alvo do salto. E pode ser associado a qualquer teste de condição.

**Exemplo:**

- No exemplo anterior (cálculo de média, utilizado para exemplificar o **break**), havia um erro. Embora seja definido que a nota máxima é dez, o programa não impede que seja informada uma nota acima de 10. Iremos corrigir isso agora, usando o “**continue**”.
- Desse modo, o laço while do exemplo anterior ficaria assim:

```

▶ while ( TotalNotas < MAX_NOTAS )
{
    cin >> NotaDaVez;
    if ( NotaDaVez < 0 ) // Não há mais notas a informar.
        break ; // Salta para a primeira instrução após o fim do laço.
    if ( NotaDaVez > 10 )
    {
        cout << "Nota não pode ser superior a 10" << endl ;
        ⬅ continue ; // Salta para o fim do laço, o que, aqui,
        // significa voltar imediatamente à linha onde é feito
        // o teste da condição de continuidade.
    }
    TotalNotas++ ;
    Soma += NotaDaVez ;
}

```

O **continue**, acima, cumpre o seu papel, desprezando todo o código abaixo.

Mas, exceto quando é necessário desprezar um trecho de código muito grande, talvez contendo muitos outros testes de decisão(e, então, um **continue** irá simplificar a lógica), devemos **dar preferência** a controles que permitam uma **única direção de fluxo** (sempre para a frente) pois isto simplifica a leitura.

*Assim, o exemplo acima ficaria melhor do seguinte modo:*

```

.....
if ( NotaDaVez > 10 )
{
    cout << "Nota não pode ser superior a 10" << endl ;
}
else // mais claro !!!
{
    TotalNotas++ ;
    Soma += NotaDaVez ;
}
.....

```



**MAS ATENÇÃO:** se estivéssemos usando um laço **for** para a situação acima, teríamos o seguinte problema:

```

for ( TotalNotas = 0 ; TotalNotas < MAX_NOTAS ; ++TotalNotas)
{
    cin >> NotaDaVez;
    if ( NotaDaVez < 0 ) // Não há mais notas a informar.
        break ; // Salta para a primeira instrução após o fim do laço.
    if ( NotaDaVez > 10 )
    {
        cout << "Nota não pode ser superior a 10" << endl ;
        ⚡ continue ; // Salta para o fim do laço, o que, aqui, significa saltar
        // para a instrução de progressão. Em seguida, executará
        // a avaliação da condição de continuidade.
    }
    Soma += NotaDaVez ;
    ⚡ } // Fim do laço, a progressão será executada agora.

```

O exemplo acima, com o **laço for**, levará a resultados **errados**.

- Isto ocorrerá se a variável **NotaDaVez**, em algum momento, apresentar um número superior a dez. Neste caso o valor será **desprezado** (devido ao **continue**).
- Mas a **próxima** coisa que será executada será a **progressão**: ++**TotalNotas**.
- E isso significa que será considerada uma **nota a mais** (e a média ficará menor).
- Usando o **laço for** não seria possível usar o **continue** desse modo. E para resolver o problema **seria preciso complicar o código**, deixando-o **mal escrito**:

```

if ( NotaDaVez > 10 )
{
    cout << "Nota não pode ser superior a 10" << endl ;
    ⚡ --TotalNotas ; // Decrementa TotalNotas, para compensar.
    // ➡ Isto resolve, mas é horrível !
    continue ; // Vai saltar para a instrução de progressão: ++TotalNotas
}

```



Observe que o problema aqui **não está no continue** e **sim** no fato de estarmos usando o **laço for**.  
Pois se usássemos o **"if ... else"** daria na mesma:

*Conforme podemos ver no código abaixo:*

```

for ( TotalNotas = 0 ; TotalNotas < MAX_NOTAS ; ++TotalNotas)
{
    .....
    if ( NotaDaVez > 10 )
    {
        cout << "Nota não pode ser superior a 10" << endl ;
        ⚡ --TotalNotas ; // Precisa decrementar TotalNotas, pois, após este
        // if-else, teremos o fim do bloco, e, em seguida, TotalNotas
        // será incrementada. Péssima solução! ☹️
    }
    else
        Soma += NotaDaVez ;
    ⚡ } // Final do laço: a progressão será executada agora ( ++TotalNotas ).

```

✎ Em determinadas situações não haverá uma diferença drástica entre usar um laço **while** ou um laço **for**.

✎ Mas, em outras ocasiões, essa escolha **fará uma grande diferença**.

**No exemplo acima**, temos uma situação em que o laço **while** é a melhor opção, representando a escolha **correta**.

E o laço **for** revelou-se **inadequado** para o caso.

✎ Assim, devemos analisar a **situação específica** para saber qual o controlador de laço **adequado** a cada uma.

E um critério importante para essa análise é:

Se usamos uma variável como contadora (determinando a progressão de um laço) *só devemos usar o laço **for** se a variável for **alterada apenas no segmento de progresso da estrutura de controle** e **não** nas instruções associadas ao laço.*

*Pois, caso isso ocorra, em geral o laço **while** será uma escolha melhor.*

#### 5.10.3.4 • Desvio **goto**.

O **goto** é um desvio minimamente estruturado. O programador **não** é livre para saltar de um ponto de uma função para outro ponto em **outra função**.

Mas, **dentro da mesma função**, o programador **é livre** para escolher o alvo do desvio. E o alvo para um desvio deve ser definido através de um **rótulo**.

✎ Já vimos mais acima um exemplo (seção 5.10.1, página 168) que mostra que isso **não é bom**. Geralmente leva a código **confuso** e/ou sujeito a **erros**.

✎ Devemos preferir o **if/else** e/ou os **laços estruturados**, conforme o caso.

## 5.11 • Revisão do Capítulo 5

### 5.11.1 • Sintetizando as regras básicas para escrita de código.

Relembrando alguns dos quesitos **básicos** para a escrita de código em C e C++:

- Criar um ou mais módulos(ou *unidades de tradução*): um **módulo** é um **arquivo texto** com extensão **“.c”** (linguagem **C**) ou **“.cpp”** (linguagem **C++**), ainda que isso seja apenas uma praxe e não uma regra da linguagem (e há quem use outras extensões para arquivos fonte **C++**, como **“.cxx”**, **“.cc”** e outras).
- Dentro de cada módulo, criar funções: uma **função** é um **bloco** onde podemos escrever instruções.
- Definir qual dessas funções representará o início ou **ponto de entrada** da aplicação. Essa função **especial e única** deverá receber o nome **main**.
- Dentro das funções, escrever **linhas de instrução**.

A figura ao lado mostra o aspecto de um **programa mínimo** usando regras comuns às linguagens **C** e **C++** (exceto **<iostream>** e **cout**, de **C++**).

- Há um único **módulo**: o arquivo **fonte.cpp**.

```
// arquivo fonte.cpp
#include <iostream>
int main() // início
{
    std::cout << "hello\n" ;
    return 0 ;
}
```

- O módulo tem uma única **função**: **main**. E, por ter esse **nome especial**, ela é o **ponto de entrada** da aplicação.
- A função tem apenas duas **linhas de instrução**:
  - uma operação de chamada a uma outra função: o **operador "<<"**, que, juntamente com o **cout**, devem fazer parte de alguma biblioteca externa, já que suas declarações não estão visíveis neste fonte;
  - **return 0** - é o retorno da função.

## 5.11.2 • Exercício 1

Tente resolver o exercício abaixo. Em seguida, compare sua solução com a que está no **Anexo B-5-1, página 446**. Caso não entenda, fale com o instrutor.

**Enunciado:** criar um projeto com dois módulos e testar várias funções.

- Neste exercício serão criados **duas unidades de tradução** (ou módulos), isto é, **dois arquivos fontes**.
  - Um deles (por exemplo, **main.cpp**) deve conter **apenas a função main**.
  - O outro (por exemplo, **funcoes.cpp**) deverá conter **diversas funções** que realizarão tarefas específicas.
- **main** deverá **testar todas as funções** implementadas no módulo **funcoes.cpp**.
- As funções que devem ser escritas no módulo **funcoes.cpp** (e que serão detalhadas abaixo) são:
  - **unsigned long long Fatorial ( unsigned int num ) ;**  
Calcula o **fatorial** de um número inteiro. **Retorna unsigned long long** porque o resultado pode não caber em um **int de 32 bits**.
  - **unsigned long long Potencia ( unsigned int base, unsigned int exp ) ;**  
Calcula o resultado: "base elevada a expoente" (apenas inteiros para simplificar).
  - **int PA\_TotalTermos ( int inicial, int final, int razao ) ;**  
Calcula o total de termos de uma Progressão Aritmética (não a sua soma).

```

Testa Fatorial:
Fatorial de 0 = 1
Fatorial de 1 = 1
Fatorial de 2 = 2
Fatorial de 3 = 6
Fatorial de 4 = 24

Testa Potencia:
10 elevado a 0 = 1
10 elevado a 1 = 10
10 elevado a 2 = 100
10 elevado a 3 = 1000
10 elevado a 4 = 10000

Testa PA_TotalTermos:
Total termos entre 1 e 10, razao 0 = 0
Total termos entre 1 e 10, razao 1 = 10
Total termos entre 1 e 10, razao 2 = 5
Total termos entre 1 e 10, razao 3 = 4
Total termos entre 10 e 1, razao -1 = 10
Total termos entre 10 e 1, razao -2 = 5
Total termos entre 10 e 1, razao -3 = 4
Total termos entre 1 e 10, razao -1 = 0
Total termos entre 10 e 1, razao 1 = 0
Total termos entre 10 e 10, razao 1 = 1
Total termos entre 10 e 10, razao -1 = 1

```

real no primeiro dia do  
dia, 4 reais no terceiro,

quando um **laço for**.

**conjunto,**  
**ed int escolhas ) ;**

n, considerando-se uma  
por exemplo, para saber

n:

tos com um **laço for**.

os de uso em que a

**Resultados (continuando a exibição):**

Nos testes  
ção apr  
- inicial  
par - e  
maior c

```
Testa ImprimePares:
Lista dos numeros pares entre 1 e 11
Total de numeros a imprimir: 5
Pares no intervalo:
2 , 4 , 6 , 8 , 10
```

```
Lista dos numeros pares entre 12 e 11
Total de numeros a imprimir: 0
Pares no intervalo:
Nenhum numero par nesse intervalo
```

**5.11.2**

```
Imprime também os intervalos: [ 1 e 10 ] [ 2 e 11 ] [ 2 e 10 ]
                                [ 10 e 10 ] [ 11 e 11 ]
```

Para co  
derá us

Este ex

```
Testa DobraValor:
Ganhando 1 real no primeiro dia de um mes,
e dobrando o valor todos os dias,
no dia 31 ganharei:
1073741824
```

Além d

```
0 mesmo calculo usando o 'for':
1073741824
```

```
Testa TotalCombinacoes:
Usando Fatorial, sei as chances de ganhar:
Na MegaSena : 1 em 50063860
Na LotoFacil: 1 em 3268759
```

- De preferência usar um *ambiente de desenvolvimento*.
- Acrescentar o arquivo, **main.cpp**, onde deverá estar a **função main**.
- **Será necessário adicionar mais um arquivo: "funcoes.cpp":**
- Tenha você usado um dos ambientes de desenvolvimento ou um editor de textos, o importante é que você tenha criado os arquivos **main.cpp** e **funcoes.cpp**.

## O que fazer no código fonte: detalhamento das funções propostas acima e dicas de implementação

- Comece pelo arquivo **funcoes.cpp**, implementando as funções:

### a. unsigned long long Fatorial ( unsigned int numero ) { ... }

Para isso, tome **como base** o código com que exemplificamos o uso do **laço "for"** para **cálculo de fatorial** (seção **3.3.2.3**, página **88**, acima).

Mas, preferencialmente, tome como base esse mesmo exemplo, modificado neste capítulo, usando **operadores compostos**: seção **5.3.4**, página **141**, acima.

### b. unsigned long long Potencia ( unsigned int base, int unsigned expoente ) { ... }

A função deve calcular o resultado de "base" elevada ao "expoente". Para simplificar, usa apenas inteiros sem sinal. Observe que a lógica dessa função será **muito semelhante à da função "Fatorial"**.



Em "**Fatorial**", temos um número **multiplicado por ele menos um**, sucessivamente, até atingir **1**. Em "**Potencia**", teremos um número (a "base") **multiplicado por ele mesmo** tantas vezes quanto indicar o "expoente". Por exemplo, se "base" for 10 e "expoente" for 2, teremos  $10 * 10$ .

#### c. `int PA_TotalTermos ( int inicial, int final, int razao ) { ... }`

O total de termos de uma Progressão Aritmética pode ser calculado simplesmente assim:

**( final-inicial+razao ) / razao ;** // atenção: **precedência** das operações...

Mas é preciso que esses valores sejam consistentes:

- **razao** não pode ser zero.
- Se **inicial** for **menor-que final**, **razao** deve ser **positiva**.
- Se **inicial** for **maior-que final**, **razao** deve ser **negativa**.
- Nesses **três casos**, a função deve **retornar zero**.

Há **duas maneiras** de implementar essas três consistências:

- Com **sucessivos** testes de condição **if else**.
- Mas pode ser **melhor resolvido** com um **único if**, usando os **operadores lógicos or e and**.

A segunda alternativa é melhor, mas, agora, o importante é que você **resolva** o problema **seja de que modo for**.

#### d. `void ImprimePares ( int inicial , int final ) { ... }`

- Deve imprimir uma mensagem inicial, informando o intervalo (inicial e final).
- Deve imprimir uma segunda mensagem informando o total de números que será impresso.
  - Para isso pode chamar a função **PA\_TotalTermos( inicial, final, 2 )**, ou seja, passando **2** como argumento para **razao**, já que essa é a distância entre os números pares.
  - Mas **atenção**: se **inicial** for **ímpar**, a função poderá não retornar o resultado esperado, uma vez que o intervalo não está correto (já que esse número **não poderá** ser impresso).
- Para imprimir todos os números pares entre "inicial e final", um **laço for** é uma boa alternativa. Mas há duas maneiras de implementá-lo:
  - Percorrer **todos os números** entre **inicial** e **final**, e, dentro do laço, antes de imprimir:
    - **Testar** se o número **é par**.
    - Se for, imprimir. Do contrário, nada a executar.
    - Em seguida o laço faz um **incremento**, passando a tratar o próximo número (se ele for menor-ou-igual a **final**).
  - Se você analisar a alternativa acima, verá que ela **duplica** o processamento.
    - Melhor seria **testar inicialmente (antes do laço) se 'inicial' é ímpar**.
    - Se for, **incrementar 'inicial'** para que se torne **par**.
    - Agora, no laço, o primeiro valor a ser impresso (se houver algum) **será um número par**.
    - E se, ao invés de incremento, **somarmos 2** para imprimir o próximo número, ele será sempre par (a distância entre números pares é **2**).
  - Nas duas alternativas acima, você terá que testar se um determinado número é par (ou ímpar). Isso é conseguido descobrindo qual é o **resto da divisão** desse número por **2**. Temos duas maneiras:
    - Usando o **operador de módulo (%)**.
    - Como queremos saber o resto da divisão por **2** (que é uma **potência de 2**) há uma maneira **mais eficiente**.

Não lembra? **Reveja este capítulo** para descobrir qual é...

Se não conseguir, use a primeira alternativa. O importante é resolver.

#### e. `double DobraValor ( int ultimo_dia ) { ... }`

Problema básico/clássico de matemática: se você ganha 1 real no primeiro dia do mês e **dobra** esse valor a cada dia (2 reais no segundo dia, 4 reais no terceiro, etc), quanto irá ganhar no último dia (por exemplo, o dia 31)?

- Essa é uma **Progressão Geométrica**, com razão de multiplicação **2** (cada termo é o **dobro** do anterior).
- A forma de cálculo do **enésimo termo** (neste exemplo, o dia 31, ou qualquer outro dia que se queira usar) é a seguinte:

[  $an = a1 * q^{(n-1)}$  ] - (modelo genérico: o símbolo  $\wedge$  simboliza exponenciação - isso **não é assim em C ou C++**)

Onde '**a1**' é o **primeiro** termo, '**q**' é a **razão** de multiplicação e '**n**' é o **total de termos**.

ou:

**an = a1 \* Potencia( q , n-1 ) ;**

Neste caso (dobrar o valor a partir do dia 1), o **primeiro** termo é o dia **1** e a razão só pode ser **2** (a cada dia ganho o dobro do dia anterior); então podemos substituir:

- '**a1**' por **1**;
- '**q**' por **2**
- e '**n**' por '**ultimo\_dia**' (que é o **parâmetro da função**).

#### f. `double DobraValor_for ( int ultimo_dia ) {...}`

Faz o mesmo usando um **laço for**.

- Dentro do laço, uma variável (previamente iniciada com **1**), será, cumulativamente, multiplicada por **2**.
- É necessária uma segunda variável, que condicione a repetição do laço, de tal modo que essa operação só seja realizada até [ **ultimo\_dia - 1** ].  
Pois, se `ultimo_dia == 1`, ganho apenas 1...

#### g. `unsigned long long TotalCombinacoes ( unsigned int conjunto, unsigned int escolhas ) { ... }`

Outro problema básico/clássico de matemática: retorna o total de combinações únicas (independentemente da ordem dos elementos), considerando-se uma quantidade de escolhas **s** como um subconjunto em um conjunto de **n** elementos:

**Fatorial ( n ) / ( Fatorial ( s ) \* Fatorial ( n - s ) )**

Por exemplo, para saber as possibilidades de acerto na mega-sena, [  $C(60,6)$  ]:

**TotalCombinacoes ( 60, 6 ) ;**

#### ■ **Teste todas as funções no arquivo `main.cpp`:**

- a. Em "**main**", onde as demais funções serão chamadas, como o compilador poderá **analisar** se essas chamadas estão sendo feitas **corretamente**?

Pois acontece que "**main**" está no arquivo "**main.cpp**" e as demais funções estão em **outro arquivo** ("**funcoes.cpp**").



Será que **protótipos de funções** e **arquivos header** resolvem isso?

Não lembra como fazer? **Reveja este capítulo** para descobrir a solução...

- b. Para alguns testes você pode usar um laço **for** (por exemplo, **Fatorial** de 0 até 4).
- c. **Sempre que se aplique**, procure usar **operadores compostos**, **operadores lógicos** e o **bitwise and**.

- d. Os testes devem explorar **todas as possibilidades de falha** de cada função (por exemplo, uma chamada a **ImprimePares**, com números ímpares). Analise os limites de cada uma e teste todas as consistências ou tratamentos que elas deveriam implementar.



**Compare o que você fez com a solução da apostila: página 446.**

### 5.11.3 • Exercício 2

**Tente** resolver o exercício abaixo. **Em seguida, compare sua solução** com a que está no **Anexo B-5-2, página 452**. **Caso não entenda**, fale com o instrutor.

**Enunciado:** Implemente uma **calculadora** que realize as 4 operações básicas (soma, subtração, divisão e multiplicação).

- Use dois módulos. O primeiro deles (**main.cpp**) deverá conter a função **main**, a qual irá testar as funções do segundo módulo. No segundo módulo (**calculadora.cpp**), escreva duas versões da função **Calculadora**.
  - As duas irão pedir ao usuário, **dentro de um laço**, que informe: o **primeiro operando, o operador e o segundo operando**.
  - Após cada operação, perguntar se deve ser feita nova operação.
  - A única diferença entre essas duas funções é que uma usará [ **if else** ] para analisar o **operador** e a segunda usará [ **switch ( operador )** ].

**void Calculadora\_if\_else ( ) ;**

**void Calculadora\_switch ( ) ;**

**Resultado que será exibido:**

```
Testa 'Calculadora_if_else'
Informe operando_1, operador e operando_2: 5 + 10
Somar: 15
Nova operacao? (0)encerrar, (1)continuar: 1
Informe operando_1, operador e operando_2: 10 * 10
Multiplicar: 100
Nova operacao? (0)encerrar, (1)continuar: 0 ← Fim

Testa 'Calculadora_switch'
Informe operando_1, operador e operando_2: 5 + 10
Somar: 15
Nova operacao? (0)encerrar, (1)continuar: 1
Informe operando_1, operador e operando_2: 10 * 10
Multiplicar: 100
Nova operacao? (0)encerrar, (1)continuar: 1
Informe operando_1, operador e operando_2: 9 % 3
operador invalido
Nova operacao? (0)encerrar, (1)continuar: 0 ← Fim
```

**Exemplo de entrada de dados e de análise do operador:**

```
std::cin >> operando_1 >> operador >> operando_2 ;
if ( operador == '+' )
```

```
std::cout << "Somar: " << operando_1 + operando_2 << '\n';
```

**Os operadores admitidos são:**

+ - \* / (soma, subtração, multiplicação, divisão).

**Se informado qualquer outro, deve ser exibida mensagem de erro:**

```
std::cout << "operador incorreto\n" ;
```

- Usando um ambiente ou o editor de textos, crie os **dois arquivos** em um **novo diretório**. Por exemplo:

```
<...>/cursoCPP/07_if_else_switch
```

- **Compilar e executar.**



**Compare o que você fez com a solução da apostila: página 452.**

### 5.11.4 • Questões para revisão

Responda às questões abaixo, assinalando **todas** as respostas corretas (**uma ou mais**). **Em seguida, compare suas respostas** com as respostas localizadas no **Anexo B-5-3, página 455**. **Caso não entenda**, encaminhe as dúvidas ao instrutor.

**Cap.5 - 1.** Assinale as afirmações verdadeiras:

- ☐ O comentário iniciado por `/*` e finalizado por `*/` só é válido em **C**.
- ☐ O comentário iniciado por `/*` e finalizado por `*/` só é válido em **C++**.
- ☐ O comentário iniciado por `/*` e finalizado por `*/` é válido em **C** e em **C++**.
- ☐ O comentário iniciado por `//` e finalizado pela quebra de linha do texto (*new-line*) só é válido em **C++** ou em **C99**, embora muitos compiladores, dependendo das opções de compilação, o aceitem para **C89**.

**Cap.5 - 2.** Em uma linha de instrução podemos:

- ☐ Encadear diversas operações, inclusive chamadas de função, independentemente dos retornos dessas funções.
- ☐ Combinar operações, inclusive chamadas de funções, desde que os retornos das funções, usadas como operandos, sejam compatíveis com os operadores associados.
- ☐ Atribuir a uma variável o retorno de uma função com valor de retorno **void**.

**Cap.5 - 3.** Assinale as afirmações corretas, considerando o código abaixo:

```
char c; for ( c = 2 ; c >= 0 ; ++c ) { std::cout << c ; }
```

- ☐ O código acima leva a um erro de compilação.
- ☐ O bloco de instruções associado ao laço será executado **2 vezes**.
- ☐ O bloco de instruções associado ao laço será executado **uma vez**.

- d. ☐ Será executado infinitamente (**laço infinito**).
- e. ☐ **Nunca** será executado.
- f. ☐ **Nenhuma** das respostas acima está correta.

**Cap.5 - 4.** Assinale as afirmações corretas, considerando o código abaixo:

```
unsigned char uc ;    for ( uc = 2 ; uc >= 0 ; ++uc )
{ std::cout << uc ; }
```

- a. ☐ O código acima leva a um erro de compilação.
- b. ☐ O bloco de instruções associado ao laço será executado **2 vezes**.
- c. ☐ O bloco de instruções associado ao laço será executado **uma vez**.
- d. ☐ Será executado infinitamente (**laço infinito**).
- e. ☐ **Nunca** será executado.
- f. ☐ **Nenhuma** das respostas acima está correta.

**Cap.5 - 5.** Assinale as afirmações corretas, considerando o código abaixo:

```
char c ; for ( c = 2 ; c >= 0 || c < 0 ; ++c ) { std::cout << c << '\n' ; }
```

- a. ☐ O código acima leva a um erro de compilação.
- b. ☐ O bloco de instruções associado ao laço será executado **2 vezes**.
- c. ☐ O bloco de instruções associado ao laço será executado **uma vez**.
- d. ☐ Será executado infinitamente (**laço infinito**).
- e. ☐ **Nunca** será executado.
- f. ☐ **Nenhuma** das respostas acima está correta.

**Cap.5 - 6.** Assinale as afirmações corretas, considerando o código abaixo:

```
int x ; for ( x=10 ; x > 1 ; --x ) { std::cout << x ; }
```

O bloco de instruções associadas ao laço for será executado:

- a. ☐ **11 vezes**.
- b. ☐ **10 vezes**.
- c. ☐ **9 vezes**, de acordo com o resultado de uma chamada à função "PA\_TotalTermos", implementada no exercício "06\_dois\_modulos", acima: [ **PA\_TotalTermos ( 10 , 2 , -1 ) ;** ]
- d. ☐ Infinitamente (**laço infinito**).
- e. ☐ **Nunca** será executado.
- f. ☐ Após o encerramento do **for** o valor de "**x**" será **zero**.
- g. ☐ Após o encerramento do **for** o valor de "**x**" será **um**.
- h. ☐ Após o encerramento do **for** o valor de "**x**" será **dois**.

**Cap.5 - 7.** Assinale as afirmações corretas, considerando o código abaixo:

```
int x ; std::cin >> x ; if ( x == 10 ) { ++x ; }
```

// atenção para o símbolo em ( x == 10 ): dois sinais de igual

Quando o 'if' acima **for executado**, a variável '**x**' será **incrementada**:

- a. ☐ 10 vezes, se 'x' for igual a 10.
- b. ☐ Sempre será incrementada uma vez.
- c. ☐ Uma vez, somente se 'x' for igual a 10.
- d. ☐ Nunca será incrementada.
- e. ☐ A operação é **incorreta** e ocorrerá um erro de compilação.
- f. ☐ Nenhuma das respostas acima está correta.

**Cap.5 - 8.** Assinale as afirmações corretas, considerando o código abaixo:

```
int x ; std::cin >> x ; if ( x = 10 ) { ++x ; }
// atenção para o símbolo em ( x = 10 ): um sinal de igual
```

Quando o 'if' acima **for executado**, a variável 'x' será **incrementada**:

- a. ☐ 10 vezes.
- b. ☐ Uma vez, somente se 'x' for igual a 10.
- c. ☐ Nunca será incrementada.
- d. ☐ Sempre será incrementada.
- e. ☐ A operação é **incorreta** e ocorrerá um erro de compilação.
- f. ☐ Nenhuma das respostas acima está correta.

**Cap.5 - 9.** Assinale as afirmações corretas, considerando o código abaixo:

```
int x ; std::cin >> x ; if ( x > 20 && x < 21 ) { ++x ; }
```

Quando o 'if' acima **for executado**, a variável 'x' será **incrementada**:

- a. ☐ Nunca será incrementada.
- b. ☐ Sempre será incrementada.
- c. ☐ Será incrementada sempre que 'x' for igual a 20.
- d. ☐ Será incrementada sempre que 'x' for igual a 21.
- e. ☐ Será incrementada se 'x' for [ maior que 20 ] OU [ menor que 21 ].
- f. ☐ A operação é **incorreta** e ocorrerá um erro de compilação.
- g. ☐ Nenhuma das respostas acima está correta.

**Cap.5 - 10.** Assinale as afirmações corretas, considerando o código abaixo:

```
int x ; std::cin >> x ; if ( x > 20 || x < 21 ) { ++x ; }
```

Quando o 'if' acima **for executado**, a variável 'x' será **incrementada**:

- a. ☐ Nunca será incrementada.
- b. ☐ Sempre será incrementada.
- c. ☐ Será incrementada **somente se 'x' for igual a 20**.
- d. ☐ Será incrementada **somente se 'x' for igual a 21**.
- e. ☐ A operação é **incorreta** e ocorrerá um erro de compilação.
- f. ☐ Nenhuma das respostas acima está correta.

**Cap.5 - 11.** Assinale as afirmações corretas, considerando o código abaixo:

```
int x ; std::cin >> x ;  
if ( ( x > 20 || x < 21 ) && ( x > 20 && x < 21 ) ) { ++x; }
```

Quando o 'if' acima **for executado**, a variável 'x' será **incrementada**:

- a. ☐ **Nunca** será incrementada.
- b. ☐ **Sempre** será incrementada.
- c. ☐ Será incrementada sempre que 'x' for **igual a 20**.
- d. ☐ Será incrementada sempre que 'x' for **igual a 21**.
- e. ☐ A operação é **incorreta** e ocorrerá um erro de compilação.
- f. ☐ Nenhuma das respostas acima está correta.

**Cap.5 - 12.** Assinale as afirmações corretas, considerando o código abaixo:

```
int x , y ; std::cin >> x >> y ;  
if ( ( x > 20 && x < 22 ) && ( y > 20 && y < 22 ) ) { ++x; }
```

Quando o 'if' acima **for executado**, a variável 'x' será **incrementada**:

- a. ☐ **Nunca** será incrementada.
- b. ☐ **Sempre** será incrementada.
- c. ☐ Sempre que 'x' for **igual a 21**; 'y' poderá conter **20, 21 e 22**.
- d. ☐ Sempre que 'y' for **igual a 21**; 'x' poderá conter **20, 21 e 22**.
- e. ☐ Sempre que [ 'x' for **igual a 21** ] **E** [ 'y' for **igual a 21** ].
- f. ☐ A operação é **incorreta** e ocorrerá um erro de compilação.
- g. ☐ Nenhuma das respostas acima está correta.

**Cap.5 - 13.** Assinale as afirmações corretas, considerando o código abaixo:

```
int x ; std::cin >> x ; while ( true ) { ++x ; }
```

A cada vez que o '**while**' acima for executado, a variável 'x' será **incrementada**:

- a. ☐ **Uma** vez.
- b. ☐ **Nunca** será incrementada.
- c. ☐ Será incrementada **infinitamente** (laço infinito).
- d. ☐ A operação é **incorreta** e ocorrerá um erro de compilação.
- e. ☐ Nenhuma das respostas acima está correta.

**Cap.5 - 14.** Assinale as afirmações corretas, considerando o código abaixo:

```
int x ; std::cin >> x ; while ( false ) { ++x ; }
```

A cada vez que o '**while**' acima for executado, a variável 'x' será **incrementada**:

- a. ☐ **Uma** vez.
- b. ☐ **Nunca** será incrementada.
- c. ☐ Será incrementada **infinitamente** (laço infinito).
- d. ☐ A operação é **incorreta** e ocorrerá um erro de compilação.

- e. ☐ Nenhuma das respostas acima está correta.

**Cap.5 - 15.** Assinale as afirmações corretas, considerando o código abaixo:

```
int y ; std::cin >> y ; int x = y > 5 && y < 7 ;
```

- a. ☐ 'x' poderá armazenar o número inteiro **5** ou o número inteiro **7**.  
 b. ☐ 'x' sempre armazenará o valor inteiro **1**.  
 c. ☐ 'x' sempre armazenará o valor inteiro **0**.  
 d. ☐ 'x' armazenará o valor inteiro **1** se 'y' for **igual a 6**.  
 e. ☐ 'x' armazenará o valor inteiro **0** se 'y' for **diferente de 6**.  
 f. ☐ A operação é **incorreta** e ocorrerá um erro de compilação.  
 g. ☐ Nenhuma das respostas acima está correta.

**Cap.5 - 16.** Assinale as afirmações corretas, considerando o código abaixo:

```
int y ; std::cin >> y ; bool x = y > 5 && y < 7 ;
```

- a. ☐ 'x' poderá armazenar o número inteiro **5** ou o número inteiro **7**.  
 b. ☐ 'x' sempre armazenará o valor *booleano* **true**.  
 c. ☐ 'x' sempre armazenará o valor *booleano* **false**.  
 d. ☐ 'x' armazenará o valor *booleano* **true** se 'y' for **igual a 6**.  
 e. ☐ 'x' armazenará o valor *booleano* **false**, se 'y' for **diferente de 6**.  
 f. ☐ A operação é **incorreta** e ocorrerá um erro de compilação.  
 g. ☐ Nenhuma das respostas acima está correta.

**Cap.5 - 17.** Assinale as afirmações corretas, considerando o código abaixo:

```
int y ; std::cin >> y ; bool x = y <= 5 || y >= 7 ;
```

- a. ☐ 'x' sempre armazenará o valor *booleano* **true**.  
 b. ☐ 'x' sempre armazenará o valor *booleano* **false**.  
 c. ☐ 'x' armazenará o valor *booleano* **true** se 'y' for **diferente de 6**.  
 d. ☐ 'x' armazenará o valor *booleano* **false**, se 'y' for **igual a 6**.  
 e. ☐ A operação é **incorreta** e ocorrerá um erro de compilação.  
 f. ☐ Nenhuma das respostas acima está correta.

**Cap.5 - 18.** Assinale as afirmações corretas, considerando o código abaixo:

```
// ...
int x ;
std::cin >> x ;
switch ( x )
{
    case 1 :
        funcao1 ( );
        break;

    case 2 :
        funcao2 ( );
```



```
    case 3 :  
        funcao3 ( );  
        return;  
    case 4 :  
        funcao4 ( );  
    default :  
        funcao5 ( );  
}  
++x ;
```

- a. ☐ se 'x' for **igual a 1**, a **funcao1** será chamada e em seguida 'x' será **incrementada**.
- b. ☐ se 'x' for **igual a 2**, a **funcao2** será chamada e em seguida 'x' será **incrementada**.
- c. ☐ se 'x' for **igual a 2**, a **funcao2** será chamada, em seguida a **funcao3** será chamada e em seguida 'x' será **incrementada**.
- d. ☐ se 'x' for **igual a 2**, a **funcao2** será chamada, em seguida a **funcao3** será chamada, ocorrendo retorno em seguida.
- e. ☐ se 'x' for **igual a 3**, a **funcao3** será chamada e em seguida 'x' será **incrementada**.
- f. ☐ se 'x' for **igual a 3**, a **funcao3** será chamada, ocorrendo retorno em seguida.
- g. ☐ se 'x' for **igual a 4**, a **funcao4** será chamada e em seguida 'x' será **incrementada**.
- h. ☐ se 'x' for **igual a 4**, a **funcao4** será chamada, em seguida a **funcao5** será chamada e em seguida 'x' será **incrementada**.
- i. ☐ a **funcao5** será chamada **apenas** quando 'x' for **maior que 4**.
- j. ☐ para qualquer valor **menor que 1** e **maior que 4** a **funcao5** será chamada, ocorrendo retorno em seguida.
- k. ☐ para qualquer valor **menor que 1** e **maior que 4** a **funcao5** será chamada, e em seguida 'x' será **incrementada**.

---

## • Capítulo 6

### ▪ Ponteiros e referências

---

6.1 • O problema do carteiro.....	187
6.2 • Trabalhando com endereços.....	189
6.2.1 • Exemplo com ponteiros.....	190
6.2.2 • Ponteiros nulos (nullptr em C++11).....	191
6.2.3 • Ponteiros para Funções.....	192
6.2.3.1 • Definindo os tipos dos ponteiros para função.....	193
6.2.3.2 • Ponteiros para função permitem estabelecer eventos.....	194
6.2.4 • Usando ponteiros como parâmetros de funções.....	195
6.2.5 • Evitando duplos acessos de memória.....	198
6.3 • Referências (lvalue).....	200
6.3.1 • Inicialização de referências.....	201
6.3.2 • Diferenciando referências de ponteiros.....	201
6.3.3 • Exemplo com referências (e parâmetros por referência).....	202
6.4 • Referências (rvalue - C++11).....	204
6.4.1 • Semântica move (C++11).....	205
6.5 • Questões para revisão do Capítulo 6.....	206

Até agora trabalhamos com variáveis e constantes nas quais armazenamos valores. Quando as declaramos, dizemos que estamos **alocando memória**. Isso significa que essas variáveis e constantes não estão em um algum lugar misterioso ou desconhecido: elas estão situadas em, e ocupam, determinadas regiões ou **locais da memória**.

E como um determinado local de memória é **acessado** pelo computador? Através do **endereço** desse local. É o que veremos melhor aqui.

## 6.1 • O problema do carteiro

Imagine uma rua, com diversas casas, que só tenha um lado, cada uma delas com uma certa quantidade de moradores. Digamos também que a **numeração** das casas evolua de acordo com o seu **tamanho**, determinado pela **quantidade de moradores**:

	Rua A				
Número:	1	3	4		8
Moradores:	Família Silva		Família Gil	Família Souza	
	João Silva	Pedro Silva	Maria Gil	José Souza	André Souza
				Marta Souza	Ana Souza
					...

Se enviarmos uma carta para "João Silva", do seguinte modo:

Para: João Silva

**Endereço: Família Silva**

São Paulo - SP - Brasil

O carteiro teria um problema. Mesmo supondo-se que a "Família Silva" fosse a única com essa denominação na cidade de São Paulo, para localizá-la seria preciso percorrer a cidade, batendo de porta em porta e perguntando se ali reside a "Família Silva". E **não adiantaria memorizar** o resultado dessa busca: nada garante que essa família não irá mudar de endereço... O endereço, salvo desastres, é fixo. Os ocupantes não.

Para o carteiro, portanto, esse seria um péssimo modo de trabalho. Para ele não interessa o **valor** que ocupa um local (isto é, os moradores). Para ele interessa o **endereço** desse local:

**Para: João Silva**

**Endereço: Rua A, número 1**

São Paulo - SP - Brasil

Nesta segunda forma, supondo-se que a "Rua A" seja única na cidade de São Paulo, o carteiro dispõe de um guia para chegar rapidamente ao local. Na realidade, os endereços precisam ser um pouco mais complexos, incluindo o Código de Endereçamento Postal, o **CEP**. Mas isto apenas reafirma a conclusão acima: o importante é ter uma forma de **endereçamento** de locais que **melhor atenda** à cada situação.



E o problema do **computador** é o mesmo do **carteiro**. Para acessar um local de memória, ele precisa do **endereço desse local**.

Podemos perfeitamente, em uma **primeira abordagem**, imaginar a memória de um computador como uma longa rua, onde cada casa tem o tamanho de **um byte**. Por exemplo, em uma máquina de **32 bits**, poderíamos ter:

Endereços	0	1	2	3	...	4.294.967.295 ou, em hexa, FFFFFFFF
Valor armazenado	?	?	?	?	...	?

Contudo, por eficiência os **bytes** podem ser agrupados em **palavras** ou **words** (uma "casa" completa). Além disso, para o programador, o modo de ver a memória que interessa está relacionado aos **tipos** empregados em sua alocação, pois eles indicam **quantos bytes** são reservados em um determinado endereço para um determinado fim.

**Por exemplo:**

**long a = 10 ; double b = 20.9 ; short c = 30 ; char d = 40 ;**

<b>Nomes identificadores</b> (puramente <b>mnemônicos</b> , interessam apenas ao <b>programador</b> )	"a"	"b"	"c"	"d"	
Endereços - hipótese - (é o que interessa para o computador)	1000	1004	1012	1014	1015
Valor armazenado	10	20.9	30	40	?

Nesse exemplo, teríamos, por hipótese, a primeira alocação de memória no endereço 1000. A partir desse endereço teríamos as demais áreas, cada qual garantindo a quantidade bytes necessária para a alocação dos respectivos tipos:

identificador	tipo	endereço inicial	tamanho	área ocupada
<b>a</b>	long	1000	4	1000 a 1003
<b>b</b>	double	1004	8	1004 a 1011
<b>c</b>	short	1012	2	1012 a 1013
<b>d</b>	char	1014	1	1014

Esse modelo pode ser implementado de diversas maneiras, dependendo do compilador e da plataforma. Mas **a essência será a mesma**: cada variável alocada tem um **endereço** (que acima chamamos de "endereço inicial") e uma garantia de reserva da quantidade de **bytes** (**tamanho**) requerida pelo seu **tipo**.

Os nomes identificadores só interessam para o **programador**: são **mnemônicos** que, na leitura do código, permitem um entendimento rápido da **finalidade** que cada área de memória desempenhará no processamento.

Já para o **computador** esses nomes não tem significado: tudo o que ele sabe (e precisa) fazer é acessar determinados **endereços**. O compilador e o *linker* (este no caso de símbolos globais) se encarregam de gerar o código de máquina necessário para que os endereços sejam resolvidos.

Desse modo, só será necessário manter um nome de símbolo no arquivo binário se uma aplicação **exportar** símbolos (funções ou variáveis globais) para outras aplicações. É o

que acontece, por exemplo, com as bibliotecas dinâmicas (DLLs no caso do *Windows* e *shared objects*, no caso de *Unix/Linux*).

Mas, para a execução pelo computador, a única maneira de acessar determinada memória (seja uma variável seja o código de uma função) é através dos endereços.

Assumindo que seja assim, em princípio isso só deveria interessar ao próprio computador e a programadores *assembler*. **Então, por que os programadores que usam C ou C++ precisam tomar conhecimento disso?**

Porque, em certas situações, trabalhar com os endereços é a melhor solução. Contudo, sempre que possível devemos escolher **referências** (que veremos abaixo) ou "**ponteiros inteligentes**" (que também veremos abaixo).

## 6.2 • Trabalhando com endereços

Como dissemos, existem muitas situações em que teremos vantagens em acessar valores através de seus endereços.

Por exemplo, se você for implementar uma árvore binária que exige constante troca de lugar entre os elementos inseridos na árvore (para que sejam mantidos sempre em ordem crescente ou decrescente), é muito mais eficiente trocar os endereços que ligam um elemento ao outro, do que trocar os valores, pois isto seria bem mais dispendioso.

Uma situação mais simples e também **muito comum** (e teremos várias situações como essa daqui para a frente): uma função que precisa **retornar mais do que um valor**. Uma forma de fazer isso é através de **parâmetros de saída**, que podem ser implementados como parâmetros que recebem **endereços ao invés de valores**.

Para isso temos os tipos **ponteiro**. Um ponteiro é uma memória que, ao invés de armazenar valores, **armazena o endereço** de um outro local de memória. E, para trabalhar com endereços, temos estes operadores:

Operador	Descrição	Exemplo
<b>&lt;tipo&gt; * &lt;p&gt;</b>	Declara um ponteiro <p>: ele poderá armazenar o endereço de memórias do tipo <tipo>.	int * p ; // 'p' poderá armazenar // o <b>endereço</b> // de objetos do tipo <b>int</b>
<b>&amp; &lt;o&gt;</b>	<b>Address of - retorna o endereço</b> de um objeto <o>.	int x ; int * p = &x ; // copia o <b>endereço</b> // de 'x' para 'p'
<b>* &lt;p&gt;</b>	Devolve o valor apontado pelo ponteiro <p>	int y = *p ; // acessa a memória // apontada por 'p' // e copia o seu valor para 'y'

Para usar os operadores acima, temos que seguir adequadamente sua lógica intrínseca:

- Endereços só podem ser armazenados em memórias **aptas a guardar endereços**.
- Pois, se o operador de endereço [ **&** ] **lê um endereço**, então precisamos também do seu oposto: um **tipo** que possa armazenar esse endereço. Para isso temos o **operador ponteiro (pointer)** representado pelo símbolo [ **\*** ].
- Assim, podemos fazer:

```
int v = 10 ;
```

```
int * p = &v ; // o operador pointer na declaração de um ponteiro 'p'.
```

- O endereço de '**v**' é **lido** pelo operador de endereço [ **&** ] (*Address of*).
- Em seguida é **gravado** em '**p**'.
- Isso é possível pois '**p**' é uma variável **capaz** de armazenar um endereço, pois foi declarada como [ **int \* p** ], o que significa que ela é um **ponteiro** (memória destinada a **armazenar endereços**).
- Agora se fizermos:

```
int v_2 = * ponteiro ; // o operador pointer em um
                        // acesso ao valor apontado
```

- Estaremos recuperando o valor apontado por '**p**'. Isto é, através dele iremos acessar o endereço de '**v**', recuperando o valor inteiro 10, que, finalmente, será atribuído a '**v\_2**'.

## 6.2.1 • Exemplo com ponteiros

### Usando ambientes ou editores de texto:

- Crie um novo projeto, em um novo diretório.
  - Por exemplo, **08\_ponteiros\_1**. O sufixo "\_1" é aí usado, pois teremos logo a seguir um segundo exemplo, onde veremos uma função que devolve mais do que um valor de retorno.
- Acrescente um arquivo. Por exemplo: **main.cpp**.
- Não é necessário digitar o código. Ele pode ser encontrado, já pronto, em:
 

```
<...>/cursoCPP/apostila/08_ponteiros_1/main.cpp.
```

### Código fonte:

```
#include <iostream>
int main()
{
    int x = 5 ; // 'x' é o apelido de um endereço de memória
                // onde armazenei o número inteiro 5;

    int * p = &x ; // 'p' é o apelido de um outro endereço de memória
                  // onde armazenei o endereço de 'x'

    std::cout << "conteudo de 'x' = " << x << '\n';
    std::cout << "endereco de 'x' = " << &x << '\n';
    std::cout << "conteudo de 'p' = " << p << '\n';
    std::cout << "endereco de 'p' = " << &p << '\n';
    std::cout << "conteudo APONTADO por 'p' = " << *p << '\n';

    return 0;
}
```

/\* Resultado:

Obs.: os endereços exibidos poderão **variar**, dependendo de plataforma e compilador. Mas a situação é **a mesma**.

```
conteudo de 'x' = 5
endereço de 'x' = 0x22ff5c
conteudo de 'p' = 0x22ff5c
endereço de 'p' = 0x22ff58
conteudo APONTADO por 'p' = 5
```

\*/

**Podemos representar graficamente esse resultado do seguinte modo:**

<b>Valor:</b>	5	<b>copia para</b> → 22ff5c
<b>Endereços:</b>	22ff5c	22ff58
<b>Mnemônicos:</b>	'x'	'p'

Na tabela acima, vemos que 'x' é um mnemônico para o endereço **22ff5c**, e, nessa posição de memória está **armazenado** o número inteiro **5**. E, quando é executada a linha:

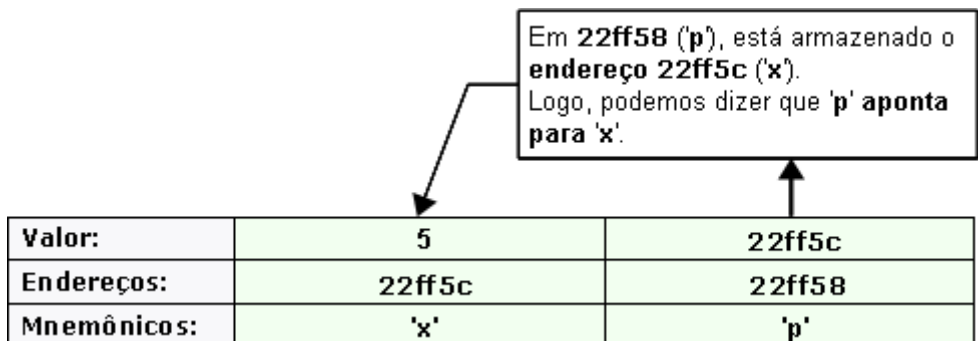
```
int * p = &x ;
```

O **endereço** de 'x' é **copiado** para 'p'. Observar que 'p' ocupa **seu próprio** local de memória (no endereço **22ff58**), constituindo assim uma segunda posição de memória em uso.

✎ Contudo, diferentemente de 'x', que armazena um **valor inteiro**, 'p' **armazena um endereço** (justamente o endereço de 'x': **22ff5c**).

Dessa forma, podemos dizer que 'p' **aponta para** 'x', já que através de 'p' podemos acessar a posição de memória 'x'.

**Essa situação pode ser visualizada na figura abaixo:**



Assim, quando é executada a linha abaixo:

```
std::cout << "conteudo APONTADO por 'p' = " << *p << "\n";
```

A operação [ **\*p** ] retorna o valor apontado por 'p' (neste caso, o valor inteiro **5**).

■ Em outras palavras, estamos ordenando:

- Acesse 'p' (no exemplo: acesse o endereço **22ff58**).
- Considere o que está **armazenado ali** como um **endereço** (no exemplo: o endereço de 'x', **22ff5c**).
- Agora **acesse esse** endereço encontrado em 'p' (no exemplo, **22ff5c**). Essa operação constitui um **redirecionamento** determinado pelo operador **ponteiro** [ **\*** ].
- Finalmente, **leia e retorne o valor** que está armazenado nesse **segundo local** (que é exatamente 'x' ou **22ff5c**).
  - Então, o valor inteiro **5** é retornado e impresso por **std::cout**.

## 6.2.2 • Ponteiros nulos (nullptr em C++11)

Um ponteiro pode ser inválido por dois motivos:

1) O ponteiro armazena um endereço de memória que não foi alocado ou que já foi liberado, por exemplo:

```
...
int *p;
*p = 10; // Erro: "p" não aponta para nenhum endereço que tenha sido já alocado
```

Neste exemplo, um erro de violação de acesso irá ocorrer, em tempo de execução, porque a variável **p** contém algum endereço ("lixo") que não foi alocado e qualquer tentativa de acesso tanto para leitura quanto para gravação vai gerar erro.

2) O ponteiro contém um **valor nulo**, por exemplo:

```
...
int *p = 0;
int *p1 = nullptr;
*p = 10; // Erro: "p" não contém um endereço válido (seu conteúdo é zero).
*p1 = 10; // Erro: "p1" também não contém um endereço válido (nullptr)
```

Tanto na linguagem C quanto em C++, **0** (zero), em um ponteiro, indica um ponteiro **nulo**.



**Contudo o 0(zero) é um int, não um ponteiro.**

Por isso, no C++11 foi introduzida a palavra reservada **nullptr** para essa finalidade, embora 0 (zero) possa continuar sendo compilado, dependendo do caso.

Mas **nullptr** deixa claro que temos um **ponteiro** com valor **nulo**, ao contrário do **zero** que é um **int**.

E também irá ocorrer erro se houver qualquer tentativa de acesso à memória através de um ponteiro que cujo conteúdo é **nullptr**.

### 6.2.3 • Ponteiros para Funções

Já vimos como chamar funções de modo convencional (invocando o seu nome e passando argumentos, se for o caso).

Mas podemos também armazenar o endereço de uma função em uma variável e chamar a função **a partir dessa variável**. Tais variáveis são denominadas como **ponteiros para funções**.

Um ponteiro para função é uma variável. E ela deverá receber o endereço de uma determinada função.

Agora, na chamada da função, iremos usar o **nome da variável** no lugar do nome da função.

Fazemos isso quando não queremos chamar uma função específica (que será sempre a mesma) e sim queremos abrir a possibilidade de que, em um determinado momento, possa haver **uma troca** da função que deve ser executada (o que será feito atribuindo o endereço de uma nova função àquela variável).

E como o compilador analisa essa chamada de função feita de modo indireto?

Sabemos que quando uma função é chamada pelo seu nome, o compilador faz o seguinte:

- a. Ele verifica se os **parâmetros** estão sendo passados de modo correto.
- b. E verifica também se não cometemos algum equívoco no aproveitamento do **valor de retorno**.



Mas, ao chamarmos uma função a partir de uma variável-ponteiro, como ficam então essas verificações que devem ser feitas pelo compilador?

### 6.2.3.1 • Definindo os tipos dos ponteiros para função.

Para que o compilador continue cumprindo seu papel de policial as chamadas de função, precisaremos **aplicar o protótipo da função a essa variável** que pretende substituir o nome da função.

E isso pode ser feito na definição do **tipo** dessa variável. Ou seja: o **tipo** de um ponteiro para função deve **representar** o próprio **protótipo** da função.

#### *Exemplo:*

*Considerada a seguinte função:*

```
int funcao ( int Param ) ;
```

**Vejamos como ela poderia ser chamada a partir de um ponteiro:**

```
int main()
{
    int ( * pFunc )(int) = funcao; // o endereço "funcao"
                                   // é atribuído à variável "pFunc"
    /* Acima, na definição do tipo da variável ponteiro, aplicou-se o
       protótipo da função:
       a) Primeiro, o retorno : int
       b) Depois, entre parênteses, fazemos a indicação de ponteiro(*) seguida
          do nome (pFunc) da variável : (* pFunc)
       c) Finalmente a lista de parâmetros: (int)

       // Agora, chamamos normalmente a função a partir da variável-ponteiro:
       pFunc ( 10 );
       return 0;
}
```

É mais interessante contudo criar os tipos **previamente** (através de **typedef**). Principalmente no caso de funções que recebem um número maior de parâmetros, essa escrita, inserida diretamente no código, prejudicará a legibilidade.

#### *Exemplo:*

*Consideradas as seguintes funções:*

```
int funcao_1 ( int Param ) ;
double funcao_2 ( int Param1 , double Param2 );
Vejamos como criar previamente os tipos:
typedef int ( * TipoPtFunc_1 ) (int) ;
typedef double ( * TipoPtFunc_2 ) ( int , double ) ;
```

*E agora utilizamos os tipos, dando mais clareza às declarações das variáveis-ponteiro:*

```
int main()
{
    // Declarar a variável a partir do tipo e atribuir o endereço da função:
    TipoPtFunc_1 pFunc_1 = funcao_1 ;

    // idem:
    TipoPtFunc_2 pFunc_2 = funcao_2 ;

    // e agora chamamos normalmente as funções a partir das variáveis:
    pFunc_1 ( 10 ) ;
    double Resultado = pFunc_2 ( 15 , 3.14 ) ;
    return 0;
}
```

}

### 6.2.3.2 • Ponteiros para função permitem estabelecer eventos.

Ponteiros para funções são muito úteis porque permitem escrever um código genérico que, nos pontos onde é necessária alguma **especificação**, pode chamar uma função a partir de um ponteiro.



Se chamamos uma função pelo seu nome sempre será executada a **mesma** ação (sempre a mesma função).



Mas se, ao invés disso, invocamos uma variável, ela poderá ser preenchida de muitas maneiras diferentes. Desse modo, para cada caso poderemos ter uma função **diferente**.

Um ponteiro para função pode assim cumprir o papel de um “*trigger*”(um disparo automático), propiciando o **disparo de eventos** sempre que alguma coisa está para acontecer ou logo após ter acontecido.

Podemos usar como exemplo uma rotina genérica para inclusão de dados em arquivos:

- Se conhecemos as regras gerais de inclusão e gravação podemos deixar o código **básico** pronto.
- Mas para que essa rotina seja realmente útil ela deve ser **flexível**, isto é, deve permitir que em cada caso específico o comportamento geral possa ser modificado.
- Por exemplo: antes que a inclusão se complete seria oportuno que uma função fosse chamada para validação final. Se tudo estivesse bem, ela retornaria “verdadeiro” e a inclusão poderia se completar. Do contrário, a inclusão seria suspensa.
- Mas se usarmos um **nome de função** não haverá flexibilidade, pois estaremos chamando sempre a mesma.
- Já se, ao contrário, usarmos uma variável, diversas aplicações específicas (cada qual utilizando arquivos diferentes, com necessidades diferentes) poderão tirar melhor proveito da nossa rotina de inclusão, pois cada uma delas poderá indicar **a sua própria função** para o momento da validação.

*Vejamos um esboço de código:*

```
// tipo para o ponteiro:
typedef bool ( * TipoPtFunc_EventoPreIncluir )( );
struct Arquivos
{
    public:
        Arquivos ( ) ; // construtora
        bool Incluir ( char * Buffer );
        bool Alterar ( char * Buffer );
        .....
        // variável ponteiro para a função "evento pré incluir":
        TipoPtFunc_EventoPreIncluir m_pfEventoPreIncluir ;
};

Arquivos::Arquivos( ) // construtora
{
    m_pfEventoPreIncluir = NULL; // Inicia ponteiro para função-evento:nulo.
    // Quando necessário, deverá ser preenchido com um endereço válido.
```

```

    // Do contrário, simplesmente não será usado
}
bool Arquivos::Incluir ( char * Buffer );
{
    // ..... // código preparatório para a inclusão
    // agora checa o ponteiro para função
    if ( m_pfEventoPreIncluir != NULL ) // se estiver nulo, é porque neste
    {                                     // caso a validação não é necessária
        if ( ! m_pfEventoPreIncluir ( ) ) // chama função a partir da variável
        {
            return false; // a função resolveu impedir que a inclusão
                          // seja finalizada
        }
    }
    // ..... // completar a inclusão
}

// A estrutura acima poderia ser usada do seguinte modo:
int main ( )
{
    Arquivos ArqClientes ;

    // preenche o ponteiro com o endereço de uma função específica:
    ArqClientes.m_pfEventoPreIncluir = OnPreIncluirClientes ;
    .....
    Arquivos ArqFornecedores ;

    // usa uma outra função para o arquivo de fornecedores:
    ArqFornecedores.m_pfEventoPreIncluir = OnPreIncluirFornecedores;
    .....
    return 0;
}

```

Exemplo adicional sobre ponteiros para funções na seção **15.6**, página **416**

## 6.2.4 • Usando ponteiros como parâmetros de funções

Há **dois motivos mais usuais** para passar argumentos por endereço para uma determinada função:

- Funções que **retornam mais do que um valor**. Uma forma de fazer isso é passar endereços, ao invés de valores.
  - Assim, através do endereço, a função poderá alterar a memória de quem a chamou.
  - Com isso, na prática, estará oferecendo mais um valor de retorno além do seu retorno formal.
  - Para que isso seja possível essas funções devem estar **aptas a receber endereços**. Ou seja, esses parâmetros "de saída" devem ser **ponteiros**.
- Uma **segunda razão é performance**, nos casos em que uma função receba um objeto (valor) de **tamanho muito grande**.
  - Se os parâmetros da função fossem objetos, haveria uma cópia da memória de quem chamou para o parâmetro da função chamada.
  - E, se esse objeto tem uma grande quantidade de **bytes**, e, pior, caso isso seja feito dentro de um laço, teremos uma grande perda de tempo copiando todos esses **bytes**.
  - Nesse caso, o correto seria passar o endereço. Pois, seja qual for o objeto envolvido, **um endereço é sempre um número inteiro sem sinal** (positivo). Desse modo, copiaremos apenas essa quantidade de **bytes**.



Veremos este **segundo** caso **no próximo capítulo**, quando trataremos de **estruturas** - e então esse problema estará colocado.

Por enquanto, vejamos um exemplo com uma **função que retorna mais do que um valor**.

#### Usando ambientes ou editores de texto:

- Crie um novo projeto, em um novo diretório. Por exemplo, **09\_ponteiros\_2**.
- Acrescente um arquivo. Por exemplo: **main.cpp**.
- Não é necessário digitar o código. Ele pode ser encontrado, já pronto, em:  
**<...>/cursoCPP/apostila/09\_ponteiros\_2/main.cpp**.

Teremos duas funções: a função **main** e a função **validar\_entrada**, que devolverá dois valores. Sua assinatura é esta:

```
bool validar_entrada( int * param_ptr ); // retorna bool;  
// e seu único parâmetro é um ponteiro para int
```

O objetivo dessa função é pegar uma entrada de dados no teclado e verificar se essa entrada é válida. Deve devolver os seguintes valores de retorno:

- O **retorno formal (bool)** indicará se a operação foi bem sucedida.
- E o **parâmetro ponteiro** devolve o **valor inteiro digitado** pelo usuário.

A função **main** irá chamar **validar\_entrada**, passando como argumento o **endereço** de uma variável sua do tipo **int**. E, **através desse endereço**, a função **validar\_entrada** irá alterar essa variável, caso a entrada de dados seja bem sucedida.

#### Resultado que deve ser exibido (ou semelhante):

```
-main: conteudo de 'x': 5  
-main: endereco de 'x': 0x22ff5c  
  
-validar_entrada: testando conteudo  
e endereco do parametro 'param_ptr'  
conteudo de 'param_ptr' = 0x22ff5c  
endereco de 'param_ptr' = 0x22ff40  
conteudo inicialmente APONTADO por 'param_ptr' = 5  
  
-validar_entrada: entre com um numero inteiro: 15 ← digitado  
  
-main: conteudo de 'x' apos 'validar_entrada' = 15
```

#### Código fonte:

```
#include <iostream>  
#include <limits>  
  
// Função que oferece dois retornos.  
// Essa função pega uma entrada no teclado.  
// - o retorno formal (bool) indica se a operação foi bem sucedida  
// - e o parâmetro ponteiro devolve o valor digitado.  
bool validar_entrada( int * param_ptr )  
{  
    // Imprime linhas para exibir conteúdo e endereço do parâmetro  
    // (apenas para teste, já que isto não tem nada a ver  
    // com o objetivo da função):
```

```

std::cout << "\n-validar_entrada: testando conteudo\n"
           << "e endereco do parametro 'param_ptr'\n";
std::cout << "conteudo de 'param_ptr' = " << param_ptr << '\n';
std::cout << "endereco de 'param_ptr' = " << &param_ptr << '\n';
std::cout << "conteudo inicialmente APONTADO por 'param_ptr' = "
           << *param_ptr << '\n';

// Agora sim, faz o trabalho real desta função:
std::cout << "\n-validar_entrada: entre com um numero inteiro: ";
std::cin >> *param_ptr ; // alterando a memória APONTADA
if ( std::cin.fail() )
{
    std::cout << "entrada invalida\n";

    // Limpa erros:
    std::cin.clear();

    // Ignora "new-lines" pendentes no buffer:
    std::cin.ignore( std::numeric_limits<int>::max() , '\n' );

    return false ; // entrada inválida
}
return true ;    // entrada válida
}

int main()
{
    int x = 5 ; // 'x' é o apelido de um endereço de memória
               // onde armazenei o NÚMERO INTEIRO 5;

    std::cout << "-main: conteudo de 'x': " << x << '\n';
    std::cout << "-main: endereco de 'x': " << &x << '\n';

    // Chama a função 'validar_entrada', passando o endereco de 'x'
    if ( validar_entrada ( &x ) )
        std::cout << "\n-main: conteudo de 'x' apos 'validar_entrada' = "
                  << x << '\n';

    return 0;
}

/* RESULTADO:

Obs.: os endereços exibidos poderão variar,
dependendo de plataforma e compilador. Mas a situação é a mesma.

-main: conteudo de 'x': 5
-main: endereco de 'x': 0x22ff5c

-validar_entrada: testando conteudo
e endereco do parametro 'param_ptr'
conteudo de 'param_ptr' = 0x22ff5c
endereco de 'param_ptr' = 0x22ff40
conteudo inicialmente APONTADO por 'param_ptr' = 5

-validar_entrada: entre com um numero inteiro: 15

-main: conteudo de 'x' apos 'validar_entrada' = 15

*/

```

Através do parâmetro **ponteiro**, **validar\_entrada** devolve o valor digitado. Foram necessários dois retornos, já que o retorno formal (**bool**) está sendo usado para indicar sucesso ou falha na operação de entrada de dados.

Quanto ao comportamento do ponteiro, exceto pelo fato de que está sendo usado como parâmetro de função, temos uma situação **idêntica** ao exemplo anterior:

Valor:	5	copia para → 22ff5c
Endereços:	22ff5c	22ff40
Mnemônicos:	'x'	'param_ptr'

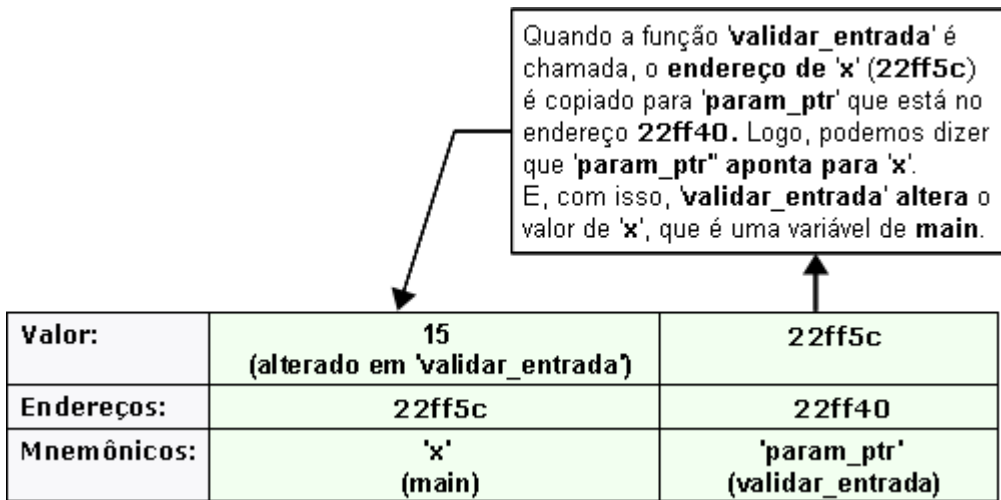
Na linha:

```
if ( validar_entrada ( &x ) )
```

O endereço de 'x' é copiado para o parâmetro ponteiro 'param\_ptr', da função 'validar\_entrada'. E, nessa função, essa memória será alterada através do ponteiro:

```
std::cin >> * param_ptr ; // alterando a memória apontada
```

Como podemos ver na figura abaixo:



Implemente algumas **variações** em um dos exercícios acima (ou ambos) .

Por exemplo, tente fazer:

```
double x = 5 ;
```

```
int * p = &x ; // o que será que acontece aqui?
```

Ou:

```
int * p = 100 ; // e aqui?
```

## 6.2.5 • Evitando duplos acessos de memória

No exemplo acima, no código necessário para atingir sua finalidade, a função **validar\_entrada** faz um único acesso à memória apontada por **param\_ptr**:


```
std::cin >> * param_ptr ;
```

Contudo há situações em que é necessário fazer muitos acessos. Isso pode comprometer a **performance**, dependendo do caso.

Pois, nos dois exemplos acima, vimos que uma operação (de leitura ou escrita) em uma memória, feita através de um ponteiro que guarde seu endereço, exige um **duplo acesso à memória**:

- **Acesse 'ponteiro'**
  - **primeiro** acesso: acessa o endereço onde está o **próprio ponteiro**.
- **Agora acesse o endereço encontrado em 'ponteiro'**
  - **segundo** acesso: acessa a **memória apontada**.

Se isso for feito muitas vezes, duplicando, a cada vez, o tempo gasto para acessos à memória, teremos problemas de **performance**.

 Há uma forma de evitar isso. É o que veremos no exemplo abaixo.

A função abaixo, deverá somar 3 ao valor inteiro apontado por '**p**', durante '**n**' vezes. Retornará **true** se o resultado for menor ou igual a 100. Do contrário, retornará **false**.

#### Exemplo 1 - acessa sempre via ponteiro:

```
bool funcao( int * p, int n )
{
    for ( int contador = 0 ; contador < n ; ++contador )
        *p += 3 ; // ... *p = *p + 3 ; // ... // duplos acessos repetidamente

    if ( *p > 100 )
        return false;
    else
        return true;
}
```

Agora essa função poderá ser chamada de muitas maneiras:

```
int main ( )
{
    int x = 5 ;

    // Na chamada abaixo, não teremos muitos duplos acessos em 'funcao':
    if ( funcao ( &x , 2 ) )
        std::cout << "- 'x' agora contem : " << x << '\n';
    else
        std::cout << "- 'x' com valor invalido apos 'funcao'\n";

    // Mas agora teremos:
    if ( funcao ( &x , 100 ) ) // 100 vezes!
        std::cout << "- 'x' agora contem : " << x << '\n';
    else
        std::cout << "- 'x' com valor invalido apos 'funcao'\n";
}
```

#### Exemplo 2 - minimizando duplos acessos:

```
bool funcao( int * p, int n )
{
    int temp = * p ; // faz uma cópia inicial do valor apontado por 'p'
                     // aqui temos um primeiro duplo acesso.
```

```

for ( int contador = 0 ; contador < n ; ++contador )
    temp += 3 ; // acumula o valor em 'temp', com um único acesso

if ( temp > 100 )
    return false ;
else
{
    *p = temp ; // Altera, de uma única vez, a memória apontada por 'p'
                // com o resultado acumulado em 'temp'.
    // Aqui temos um segundo duplo acesso - e não mais que isso.

    return true;
}
}

```

Nesta segunda versão, se a função for chamada com um valor alto para 'n', não teremos problemas. Pois o laço já não está fazendo duplos acessos.

Então, em uma chamada como [ **funcao ( &x , 100 )** ], que acarretaria 100 duplos acessos (totalizando 200 acessos) na primeira versão, teremos **agora** apenas 100 acessos a 'temp'. O duplo acesso [ \*p ] é feito **apenas duas vezes**: no início e no final.

---

 Em **C**, essa seria a **única** alternativa para minimizar os duplos acessos. Mas, em **C++**, temos uma alternativa **melhor**: usar **referências**.

---

## 6.3 • Referências (lvalue)

Referências (que só existem em C++), **resolvem, quando possível e adequado, o problema do duplo acesso**, sem que o programador precise preocupar-se com isso. Além disso referências **não** podem ser **corrompidas** (ao contrário de ponteiros).

- Um ponteiro é uma **nova** memória (ocupando um novo endereço) e que visa a **armazenar o endereço** de uma **outra** memória.
- Uma **referência** é apenas um **nome alternativo (um sinônimo)** para um endereço de memória já existente.

A criação desse **sinônimo** é conseguida através do **operador de referência**:


```

int a ;
int &ra = a ; // 'ra' é uma referência (um sinônimo) para 'a'.

```

Em outras palavras, se, por exemplo, a variável "a" for um apelido para o endereço 1000 da memória, então a variável "ra" será apenas um segundo apelido para **esse mesmo endereço** (1000).

---

 Isso significa que uma instrução que **declara uma referência**, como  
**int &ra = a ;**  
 não cria uma nova memória e sim apenas um novo nome para uma **memória já existente**.

---

Podemos então ter **funções** com **parâmetros referência**, ao invés de ponteiros. Nada será preciso fazer para minimizar duplos acessos.

**Para o programador**, esse parâmetro será simplesmente um sinônimo para o argumento que será passado na chamada da função.



**Detalhe.**

Não importa aqui como a implementação de cada compilador irá resolverá o problema. Sabemos que, na passagem de argumentos, não é possível implementar um "sinônimo". A implementação mais provável é que o argumento seja passado por endereço e, dentro da função, dependendo do custo-benefício, o compilador implemente um código semelhante ao que usamos acima no "**Exemplo 2** -minimizando duplos acessos".

Além disso, referências trazem **outros benefícios** muito importantes, como veremos mais a frente em alguns tópicos como operadores, *templates* e outros.

E um desses benefícios é tornar o código **mais limpo, legível e fácil de manter**.

### 6.3.1 • Inicialização de referências.

**Regra básica:**

Variáveis-Referência devem **obrigatoriamente** ser **inicializadas**, pois **só podemos referenciar algo já existente**.

Assim, a referência já deve nascer associada àquela memória que ela pretende referenciar.

**Exemplos:**

```
int a ;
```

```
int &a = a ; // 👍 Correto! 'ra' foi inicializada.
```

```
int &rb ; // ☹️ Errado! 'rb' não foi inicializada. Erro de compilação.
```

### 6.3.2 • Diferenciando referências de ponteiros.

Embora possa parecer confuso que o operador **&(referência)** tenha o mesmo **símbolo** do operador **&("address of")**, na prática não há confusão pois eles sempre aparecem em **posições diferentes** e assim o compilador tem como analisá-los corretamente.



Quando se trata de **&("address of")** o operador aparece sempre na posição à direita, de **captura (leitura) de um endereço**:

**Endereço:**

```
int a = 5 ;
```

```
int * pa = &a ; // Aqui, o operador & captura o endereço de a (leitura).  
// Em seguida, o operador de atribuição [=]  
// armazena esse endereço em pa (gravação);
```



Quando se trata de **&("referência")** o operador aparece sempre na posição à esquerda, de **recepção ou associação**:

**Referência:**

```
int a = 5 ;
```

```
int &a = a ; // Aqui o operador & faz com que ra referencie a  
// (tornando-se um um sinônimo). Pois o símbolo &  
// não está na posição de leitura, e sim na posição de  
// recepção ou associação, à esquerda.
```

### 6.3.3 ▪ Exemplo com referências (e parâmetros por referência)

Vamos usar, em um único exemplo, um código muito semelhante ao que vimos nos dois exemplos com ponteiros. Teremos o uso de referências dentro de uma função (**main**) e o uso de referências como parâmetros de funções.

#### Usando ambientes ou editores de texto:

- Crie um novo projeto, em um novo diretório. Por exemplo, **10\_referencias**.
- Acrescente um arquivo. Por exemplo: **main.cpp**.
- Não é necessário digitar o código. Ele pode ser encontrado, já pronto, em:  
`<...>/cursoCPP/apostila/10_referencias/main.cpp`.

#### Resultado que deve ser impresso (ou semelhante):

```
conteudo de 'x' = 5
endereco de 'x' = 0x22ff58
conteudo de 'r' = 5
endereco de 'r' = 0x22ff58
conteudo de 'p' = 0x22ff58
endereco de 'p' = 0x22ff54
conteudo APONTADO por 'p' = 5

-validar_entrada: testando conteudo
e endereco do parametro referencia 'param_ref'
conteudo inicial de 'param_ref' = 5
endereço de 'param_ref' = 0x22ff58

-validar_entrada: entre com um numero inteiro: 20 ← digitado
-main: conteudo de 'x' apos 'validar_entrada' = 20
```

#### Código Fonte:

```
#include <iostream>
#include <limits>

// Função que oferece dois retornos. Essa função pega uma entrada no teclado.
// - o retorno formal (bool) indica se a operação foi bem sucedida
// - e o parâmetro referência devolve o valor digitado.
bool validar_entrada( int & param_ref ) // 'param_ref' é uma referência
{
    // Imprime linhas para exibir conteúdo e endereço do parâmetro
    // (apenas para teste, já que isto não tem nada a ver
    // com o objetivo da função):
    std::cout << "\n-validar_entrada: testando conteudo\n"
               << "e endereco do parametro referencia 'param_ref'\n";
    std::cout << "conteudo inicial de 'param_ref' = " << param_ref
               << '\n';
    std::cout << "endereço de 'param_ref' = " << &param_ref << '\n';

    // Agora sim, faz o trabalho real desta função:
    std::cout << "\n-validar_entrada: entre com um numero inteiro: ";
    std::cin >> param_ref ; // alterando a memória referenciada
    if ( std::cin.fail() )
    {
        std::cout << "entrada invalida\n";
        // Limpa erros:
        std::cin.clear();
    }
}
```

```

    // Ignora "newlines" pendentes no buffer:
    std::cin.ignore( std::numeric_limits<int>::max(), '\n' );

    return false ; // entrada inválida
}

return true ; // entrada válida
}

int main()
{
    int x = 5 ; // 'x' é o apelido de um endereço de memória
                // onde armazenei o número inteiro 5;

    int * p = &x ; // 'p' é o apelido de um outro endereço de memória
                  // onde armazenei o endereço de 'x'

    int & r = x ; // 'r' é uma referência para 'x': ou seja, um novo apelido
                  // (um sinônimo) para a mesma posição de memória
                  // já apelidada anteriormente como 'x'

    std::cout << "conteudo de 'x' = " << x << '\n';
    std::cout << "endereço de 'x' = " << &x << '\n';
    std::cout << "conteudo de 'r' = " << r << '\n';
    std::cout << "endereço de 'r' = " << &r << '\n';
    std::cout << "conteudo de 'p' = " << p << '\n';
    std::cout << "endereço de 'p' = " << &p << '\n';
    std::cout << "conteudo APONTADO por 'p' = " << *p << '\n';

    // Chama a função 'validar_entrada', passando uma referência para 'x'
    if ( validar_entrada ( x ) )
        std::cout << "\n-main: conteudo de 'x' apos 'validar_entrada' = "
                    << x << '\n';

    return 0;
}

/* RESULTADO:

    conteudo de 'x' = 5
    endereço de 'x' = 0x22ff58
    conteudo de 'r' = 5
    endereço de 'r' = 0x22ff58
    conteudo de 'p' = 0x22ff58
    endereço de 'p' = 0x22ff54
    conteudo APONTADO por 'p' = 5

    -validar_entrada: testando conteudo
    e endereço do parametro referencia 'param_ref'

    conteudo inicial de 'param_ref' = 5
    endereço de 'param_ref' = 0x22ff58

    -validar_entrada: entre com um numero inteiro: 20

    -main: conteudo de 'x' apos 'validar_entrada' = 20

*/

```

A figura abaixo ilustra a situação de memória obtida por essa execução:

Quando 'r' é criada, temos uma referência para 'x'. Observar que o endereço de 'x' e de 'r' é exatamente o mesmo: 22ff58.  
Quando a função 'validar\_entrada' é chamada, uma referência para 'x' é criada ('param\_ref'). Observar que o endereço de 'x', de 'r' e de 'param\_ref' é exatamente o mesmo: 22ff58.  
E, através do parâmetro referência('param\_ref'), 'validar\_entrada' altera o valor de 'x', que é uma variável de main.  
**Já o ponteiro 'p'** ocupa uma **outra** posição de memória (endereço 22ff54) e **aponta para 'x'**, já que armazena o seu endereço.

Valor:	20 (alterado em 'validar_entrada')	22ff58
Endereços:	22ff58	22ff54
Mnemônicos:	'x', 'r' e 'param_ref' ( 'x' e 'r': main ( 'param_ref': validar_entrada)	'p' ( <u>main</u> )

### 6.4 • Referências (rvalue - C++11)

Antes de começarmos a falar sobre referências rvalue, devemos entender o que são **lvalues** (left values) e **rvalues** (right values). Um **lvalue** é tudo aquilo que pode receber uma atribuição como, por exemplo: `int a = 5`; no caso, “a” é o valor que está à esquerda da operação e “a”, por ser uma variável, pode receber uma atribuição, portanto “a” sempre será um **lvalue** independente da posição em que esteja de uma atribuição, seja do lado direito ou esquerdo. O valor **5** é um **rvalue**, pois **5** é uma memória **temporária** (literal) que esta a direita da operação e nunca poderia ficar à esquerda porque não podemos atribuir valores em um literal.

A referência **rvalue** pode referenciar uma memória **temporária**, ao contrário da referência tradicional que só pode referenciar um **lvalue**.

As regras para as referências rvalues continuam sendo as mesmas que para as lvalues, ou seja, não podem referenciar outra região de memória e devem sempre ser inicializadas.

A referência rvalue tem como principal objetivo realizar a semântica **move**, que permite mover (**transferir**) o estado de objetos.

```
#include <iostream>
```

```
// Passagem de parâmetros por referência lvalue:
```

```
int soma(int &a, int &b)
{
    return a + b;
}
```

```
//Passagem de parametros por referencia rvalue:
```

```
int somaComRvalue(int &&a, int &&b)
{
    return a + b;
}
```

```
int main()
{
    int a = 10;
    int &b = a; // Referencia lvalue
    // int &c = 10; // Erro, não pode referenciar um rvalue (10, que é temporário)
    int &&d = 10; // OK: "d" é rvalue, logo pode referenciar 10 (que é temporário: rvalue)
    // std::cout << soma(10, 10) << '\n'; // Erro: tentando passar temporários para lvalues...
    std::cout << "Soma Lvalue: " << soma(a, b) << '\n'; // Correto: "a" e "b"
                                                    // não temporários
    std::cout << "Soma Rvalue: " << somaComRvalue(20, 20) << '\n'; // Ok: rvalues.
    return 0;
}
```

### 6.4.1 • Semântica move (C++11)

Através da semântica move, o estado de um objeto pode ser **movido** a outro, **ao invés de ser copiado**. Isso pode trazer grandes ganhos de **performance** quando os objetos consomem grandes quantidades de memória alocadas dinamicamente.

Se possuímos um vetor, copiá-lo, significa alocar uma nova região de memória do mesmo tamanho e copiar elemento a elemento para essa nova região de memória. Mover, significa que podemos pegar o ponteiro do objeto armazenado pela origem e mover para o destino, não sendo necessário a alocação de uma nova memória e nem a cópia individual de cada elemento.



Mas lembre que:

- **copiar** significa criar ou alterar um segundo objeto a partir de um primeiro (que continua **inalterado**)
- **mover** significa **transferir** o conteúdo de um objeto para outro (e o primeiro não mantém o seu valor, tornando-se vazio ou **nulo**).

```
#include <iostream>
#include <vector>
#include <string>

void push_back_default()
{
    std::string str = "Ola mundo";
    std::vector<std::string> v;
    std::cout << "str antes push_back default: " << str << '\n';
    v.push_back(str);
    std::cout << "str depois push_back default: " << str << '\n'; // "str" manteve o seu valor

    std::cout << "v[0]: " << v[0] << "\n\n"; // o valor de "str" foi copiado para "v[0]"
}

void push_back_move()
{
    std::string str = "Ola mundo";
    std::vector<std::string> v;
    cout << "str antes push_back move: " << str << '\n';
    v.push_back(std::move(str)); // Chama o push_back que recebe um rvalue
    cout << "str depois push_back move: " << str << '\n'; // ➔ "str" foi movido;
                                                    // não manteve o seu valor (agora é nulo)
```

```
cout << "v[0]: " << v[0] << "\n\n"; // ➔ "v[0]" agora contem o antigo valor de "str"
}

int main()
{
    push_back_default();
    push_back_move();

    return 0;
}

/* Resultado:
str antes push_back default: Ola mundo
str depois push_back default: Ola mundo // "str" manteve seu valor!
v[0]: Ola mundo // "v[0]" contem uma cópia de "str"

str antes push_back move: Ola mundo
str depois push_back move: // "str" não manteve seu valor; "vazio" (ou nulo)!
v[0]: Ola mundo // o conteúdo de "str" foi transferido para "v[0]"
*/
```

## 6.5 • Questões para revisão do Capítulo 6

Responda às questões abaixo, assinalando **todas** as respostas corretas (**uma ou mais**). **Em seguida, compare suas respostas** com as respostas localizadas no **Anexo B-6, página 468**. **Caso não entenda**, encaminhe as dúvidas ao instrutor.

**Cap.6 - 1.** Assinale as afirmações corretas, considerando o código abaixo:

```
int x = 5;
int * px = &x;
```

- a. ☐ 'px' é uma **referência** para 'x'.
- b. ☐ 'px' é um **ponteiro** para 'x'.
- c. ☐ 'px' guarda o endereço de 'x'.
- d. ☐ O valor armazenado em 'px' é 5.
- e. ☐ O valor armazenado em 'x' é 5.
- f. ☐ É correto fazer: **\*x = 10;**
- g. ☐ É correto fazer: **\*px = 10;**

**Cap.6 - 2.** Assinale as afirmações corretas, considerando o código abaixo:

```
int x = 5 ;
int & rx = x ;
```

- a. ☐ 'rx' é uma **referência** para 'x'.
- b. ☐ 'rx' é um **ponteiro** para 'x'.
- c. ☐ 'rx' guarda o endereço de 'x'.
- d. ☐ O valor armazenado em 'rx' é 5.
- e. ☐ O valor armazenado em 'x' é 5.
- f. ☐ 'rx' é um **sinônimo** de 'x'.

- 
- g. ☐ É correto fazer: **\*rx = 10;**
- 
- h. ☐ É correto fazer: **x = 10;**
- 
- i. ☐ É correto fazer: **rx = 10;**
- 

**Cap.6 - 3.** Assinale as afirmações corretas, considerando o código abaixo:

```
int x = 5 ;  
int * px ;  
px = &x ;
```

- 
- a. ☐ 'px' é um **ponteiro** para 'x'.
- 
- b. ☐ O valor **apontado** por 'px', a partir da terceira linha, é **5**.
- 
- c. ☐ O código acima está incorreto e ocorrerá um erro de compilação.
- 

**Cap.6 - 4.** Assinale as afirmações corretas, considerando o código abaixo:

```
int x = 5 ;  
int & rx ;  
rx = x ;
```

- 
- a. ☐ 'rx' é uma **referência** para 'x'.
- 
- b. ☐ O valor armazenado em 'rx', (o mesmo que "x") a partir da terceira linha, é **5**.
- 
- c. ☐ O código acima está incorreto e ocorrerá um erro de compilação.
-

## • Capítulo 7

### ▪ Iniciando orientação a objetos

7.1 • Estruturas.....	210
7.1.1 • Estruturas e funções relacionadas.....	212
7.1.2 • Limitação da linguagem C.....	212
7.1.3 • Passando estruturas como argumento.....	214
7.1.4 • A especificação <i>const</i> .....	217
7.2 • Estruturas em C++ : encapsulamento.....	217
7.3 • Implementando as funções membras.....	219
7.4 • As palavras reservadas <i>struct</i> e <i>class</i> .....	221
7.4.1 • Definindo o objeto apontado por “ <i>this</i> ” como <i>const</i> .....	222
7.4.2 • Implementação das funções membras <i>const</i> .....	223
7.4.3 • Testando a <i>class Data</i> (testadata.cpp).....	229
7.4.4 • Especificando funções como <i>inline</i> .....	230
7.4.5 • Usando um segundo parâmetro do tipo “ <i>Data</i> ”.....	231
7.4.6 • Acrescentando funções operadoras à classe “ <i>Data</i> ”.....	233
7.4.7 • Os operadores << e >> de ostream/istream.....	235
7.4.8 • Conclusão Parcial.....	236
7.5 • Encapsulamento: reforçando conceitos.....	237
7.5.1 • Funções inline.....	237
7.5.2 • O ponteiro <i>this</i> .....	237
7.5.2.1 • Definindo o objeto apontado por <i>this</i> como <i>const</i> .....	238
7.5.2.2 • Exceção para a restrição <i>const</i> de <i>this</i> ( <i>mutable</i> ).....	239
7.5.3 • Função Construtora.....	239
7.5.3.1 • Parâmetros para a função construtora.....	240
7.5.3.2 • Construtora <i>default</i> .....	240
7.5.3.3 • Construtora de cópia e operador de atribuição.....	241
7.5.3.4 • Construtoras com conversão implícita e explícita.....	242
7.5.3.5 • Delegando construtoras (C++11).....	243
7.5.4 • Função destrutora.....	244
7.5.5 • Alterando o ponto de entrada da aplicação: objetos externos.....	245
7.5.6 • Classes e funções amigas.....	245
7.5.7 • Membros estáticos de uma classe.....	246
7.5.7.1 • Membros de dados estáticos.....	246
7.5.7.2 • Funções-membro estáticas.....	248
7.5.8 • Objetos como membros de classes.....	249
7.5.8.1 • Membros “objeto de outra classe”.....	249
7.5.8.2 • Membros “ponteiro para objeto de outra classe”.....	250
7.5.8.3 • Membros “ponteiro para objeto da mesma classe”.....	251
7.5.9 • Literais definidos pelo usuário (C++11).....	252
7.5.10 • Ambientes de nomes.....	253
7.5.10.1 • Evitando colisões de nomes.....	253
7.5.10.2 • Usando namespace para organizar conjuntos de software.....	254
7.5.10.3 • O namespace “std”.....	255
7.5.10.4 • Namespace inline (C++11).....	256
7.5.10.5 • Exemplos de uso de namespace.....	257
7.6 • Questões para revisão do capítulo 7.....	260



Orientação a objetos é uma das técnicas de programação suportadas por C++. Em C, é possível implementar essa técnica, mas ao custo de um trabalho extra considerável, já que ela não é suportada diretamente (nativamente) pela linguagem.

Resumidamente, essa técnica é baseada em três princípios:

- **Encapsulamento:** recursos para **proteger os dados com o código**, de maneira **inviolável**.
  - Pois muitos dos dados que usamos em um programa de computador devem obedecer a determinadas **regras**.
  - Por exemplo, não existe o dia 32. Há uma regra para isso.
  - Então, uma variável "**dia**" deveria sempre estar "escondida", isto é, **não acessível** para o conjunto do programa.
  - Só deveria ser acessada em **funções que conhecem a regra** e verificam se ela foi **violada**.
  - E, ocorrendo uma violação, uma condição de erro deve ser estabelecida.
- **Herança:** recursos para **reaproveitamento de código**.
  - Uma determinada camada de código realiza determinada tarefa.
  - Para certas situações, é necessário **complementar** (ou **estender**) as características e/ou as funcionalidades envolvidas por essa tarefa.
  - Nesse caso podemos criar uma **segunda** camada de código que deverá **herdar** tudo o que já foi feito na camada superior, acrescentando o que for necessário.
  - Isso deve ser feito de tal modo que o código já pronto (a primeira camada) **não sofra alterações**, de tal modo que aquilo que já está pronto continue funcionando sem correr o risco de que erros sejam introduzidos em uma alteração.
  - Não haverá também a necessidade de testar novamente o código anterior, já que não foi alterado.
- **Polimorfismo:** aqui temos também recursos para **reaproveitamento de código**, em um nível mais genérico.
  - Em vários casos, ao escrevermos uma camada de código básica, sabemos que ela pode implementar uma série de funcionalidades necessárias, mas não todas.
  - Para que aquilo que é reaproveitável seja útil, é preciso que essa camada não pretenda ser completa.
  - Ao contrário, ela deve **deixar em aberto** determinadas funções que devem ser executadas, mas podem ser executadas de **diferentes formas** (polimorfismo = múltiplas formas).
  - Por exemplo: um **agendador de tarefas**. Podemos construir uma aplicação que tem a capacidade de medir o tempo e saber que, em determinado tempo, determinada tarefa deve ser executada.
  - Se o agendador quisesse fazer tudo, ele executaria **apenas** tarefas bem conhecias no momento em que foi desenvolvido. No futuro, surgindo novas tarefas que também precisem ser agendadas, o nosso agendador não serviria para tratá-las.
  - Assim, o agendador deve resumir-se àquilo que é genérico: medir o tempo e ter a abertura para, no devido momento, executar **qualquer tarefa**.
  - Então, em outras camadas de código, podemos implementar tarefas específicas (como efetuar um **backup** ou atualizar um **anti-vírus**, etc, etc.).
  - Cada uma dessas **tarefas específicas** poderá utilizar os serviços do **agendador de tarefas genérico**, para que ela seja posta em execução no momento adequado.

Neste capítulo vamos tratar dos **aspectos mais importantes de encapsulamento**. Pois esse é o princípio mais importante, sem o qual os demais não fariam muito sentido, já que não estaria garantida a integridade dos dados.

**Encapsulamento** é pois a base (o **alicerce**) para implementações sólidas de orientação a objetos.

Para sua implementação em **C++**, o ponto de partida é um recurso que já existia em **C**, embora de modo incompleto: as **estruturas** de dados.

## 7.1 • Estruturas

Quase sempre temos dados que se **complementam** (podendo até ser interdependentes) e que, assim, formam um **conjunto de dados**. Precisamos de uma maneira de expressar esse fato real no código.

**Por exemplo:**

- Uma data, é composta (pelo menos) de 3 informações:
  - dia
  - mês
  - ano
- Essas informações são **interdependentes**. Dependendo do **mês**, o último dia é 30 ou 31. E, para o mês de fevereiro, dependendo do **ano** o último dia será 28 ou 29.
- Além disso, mesmo que não houvesse essa interdependência, sabemos que esses dados atuam em **conjunto** (complementares) para estabelecer uma **data completa**.
- E existem **regras**: regras para dia, regras para mês (entre 1 e 12) e regras que podem ser estabelecidas para o ano, dependendo da finalidade de uma data.

Em **C**, poderíamos organizar esse conjunto de dados da seguinte forma:

```
struct Data
{
    int dia ;
    int mes ;
    int ano ;
};
```

Uma estrutura (**struct**) permite agrupar informações que formam um conjunto. Ela é apenas um **modelo** para um **novo tipo de dados**. Esse tipo é constituído de **campos ou membros** (cada um dos dados agrupados).

Esse novo tipo, pode ser usado para criar variáveis, as quais serão alocadas na memória de acordo com o modelo declarado pela **struct**.

**Por exemplo:**

```
// Declarando variáveis de diversos tipos:
long x ;
double d ;
struct Data data_pagamento ;
struct Data data_vencimento ;
```

**Detalhando:**

Tipo	Variável	Comentário
<b>long</b>	x	'x' é uma variável do tipo <b>long</b>
<b>double</b>	d	'd' é uma variável do tipo <b>double</b>
<b>struct Data</b>	data_pagamento	'data_pagamento' é uma variável do tipo <b>struct Data</b>

<b>struct Data</b>	data_vencimento	'data_vencimento' é uma variável do tipo <b>struct Data</b>
--------------------	-----------------	---

Após as declarações acima, podemos ilustrar esquematicamente a situação da **memória** com a seguinte tabela:

<b>endereço (hipótese)</b>	<b>1000</b>	<b>1004</b> (1000+4)	<b>1012</b> (1004+8)	<b>1024</b> (1012+12)	<b>1036</b> (1024 + 12)
<b>mnemônico</b>	<b>x</b>	<b>d</b>	<b>data_pagamento</b>	<b>data_vencimento</b>	
<b>composição</b>	<b>um único valor</b> (4 bytes)	<b>um único valor</b> (8 bytes)	<b>dia</b>   <b>mes</b>   <b>ano</b> (4+4+4 = 12 bytes em SO de 32 bits)	<b>dia</b>   <b>mes</b>   <b>ano</b> (4+4+4 = 12 bytes em SO de 32 bits)	

**Atribuindo valores a essas variáveis:**

```
x = 9 ; // 'x' é um único valor (apenas um campo de informação)
d = 8.7 ; // idem para 'd'
```

Mas, ao contrário de 'x' e 'd', 'data\_pagamento' e 'data\_vencimento' não são constituídas por um único valor, sendo, ao contrário **um conjunto com 3 campos** de informação. Então deve haver um meio de **acessar cada campo**. Isso é feito com um **ponto**.

**Acessando cada campo das variáveis estruturadas** (o **ponto** acessa o campo):

```
data_pagamento.dia = 10 ; // acessa o campo 'dia' (pagamento)
data_pagamento.mes = 2 ; // idem para 'mes'
data_pagamento.ano = 2010 ; // e para 'ano'

data_vencimento.dia = 5 ; // acessa o campo 'dia' (vencimento)
data_vencimento.mes = 3 ; // idem para 'mes'
data_vencimento.ano = 2011 ; // e para 'ano'
```

**E agora teremos na memória:**

variável	x	d	data_pagamento			data_vencimento		
			dia	mes	ano	dia	mes	ano
valor	9	8.7	10	2	2010	5	3	2011

**Sintetizando:**

- Uma **struct** funciona como um descritor de campos. A estrutura é assim um **conjunto de campos**.
  - É algo muito **semelhante** à criação de uma tabela em um **banco de dados**.
  - Quando criamos uma nova tabela em um banco de dados, o que fazemos é justamente definir os **nomes e tipos** dos diferentes **campos**.
- Para a linguagem, uma **struct** cria um novo **tipo de dados**. Assim, após declarar uma **struct** poderemos criar variáveis desse **tipo**.
- E para acessar os dados internos da **struct** (isto é, seus **campos** ou **membros**):
  - usamos: o **operador de ligação de membro**, cujo símbolo é um **ponto**:

```
data_pagamento.dia = 10 ;
// a ligação do membro (ou campo) é simbolizada por um ponto
```

### 7.1.1 • Estruturas e funções relacionadas

É possível perceber, no exemplo acima, a utilidade da **struct**. O simples fato de agrupar os campos, permite expressar no código uma situação real: esses dados devem andar sempre juntos, em um único conjunto. Mas isso **ainda não é suficiente**.

Após definir conjuntos de dados (**structs**), passamos a precisar de **funções especializadas no tratamentos desses dados**. Particularmente, é preciso garantir que sejam aplicadas as **regras** que devem regular os valores possíveis para cada campo.

Assim precisamos, **no mínimo**, de funções que **validem** os campos. Além disso outras funções quase sempre são necessárias para atuar sobre cada conjunto de dados: imprimir, realizar cálculos, etc.

No caso da **struct data**, além de **funções que garantam valores válidos** para dia, mes e ano, precisamos também de **outras funções** que implementem o tratamento de todas as necessidades relacionadas a uma data.

Como, por exemplo, avaliar se uma data é maior que outra; ou calcular a diferença entre duas datas em número de dias, etc.

### 7.1.2 • Limitação da linguagem C

Na linguagem C, também escrevemos funções para atuar sobre cada **struct**.

Contudo, em C, não há uma maneira de relacionar intimamente os dados com as funções. Para a linguagem essas são funções como outras quaisquer. Do ponto de vista do programador determinadas funções foram feitas para trabalhar com determinada **struct**. Mas, em C, não há como expressar essa necessidade real no código.



A consequência disso é que, em **C**, **não** podemos **garantir** (exceto com um trabalho extra considerável) que os **dados só serão alterados em funções que conhecem suas regras e impedem sua violação**.  
Ou seja: **qualquer função** poderá alterar os campos de uma **struct**.

#### Exemplo:

Poderíamos implementar a seguinte função, que recebendo uma data como argumento, altera os seus campos. Mas **analisa** se os dados enviados para alteração estão em conformidade com as regras.

```
struct Data
{
    int dia ;
    int mes ;
    int ano ;
    int ok ; // flag que indica se a data está correta ou não, caso
            // contenha valores incompatíveis com as regras
            // (o seu tipo é int, pois, em C, não existe o bool)
};
```

```

struct Data data_altera ( struct Data dt , int dia , int mes , int ano )
{
    dt.dia = dia ;
    dt.mes = mes ;
    dt.ano = ano ;
    if ( < avaliação > ) // avalia se dia, mes e ano estão em
                        // conformidade com as regras.
    {
        dt.ok = 1 ; // os valores estão corretos
    }
    else
    {
        dt.ok = 0 ; // condição de erro
    }
    return dt ;
}

```

E o *flag* 'ok' poderá ser usado em qualquer outra função que faça a leitura dos dados de uma data:

```

void data_imprime ( struct Data dt )
{
    if ( dt.ok )
    {
        // os valores estão corretos: imprime a data normalmente
    }
    else
    {
        // condição de erro: imprime mensagem de erro.
    }
}

```

Além disso seria preciso garantir que a data **não será impressa sem que tenha sido anteriormente alterada**, de modo que o campo **ok** contenha o resultado de uma avaliação, e não "lixo" (um valor arbitrário qualquer).



Para isso deveria haver uma função **inicializadora**.

A inicialização garante um estado estável para uma estrutura desde o seu nascimento. Por exemplo, estabelecendo a condição de erro [ **dt.ok = 0 ;** ]. Ou utilizando valores *default* (caso existam).

Em suma: **inicialização** é um elemento **indispensável** quando falamos em **encapsulamento**.

Contudo, mesmo que escrevêssemos uma função "**data\_inicia**":

```

struct Data data_inicia ( struct Data dt )
{
    // ...
    dt.ok = 0 ; // condição de erro
    return dt ;
}

```

Não há como garantir que ela sempre será chamada **imediatamente após** a criação de uma variável do tipo **struct Data**.

Tudo dependerá do programador. Se o programador **nunca esquecer** de chamar as funções adequadas, tudo correrá bem. Mas isso não é uma base segura para escrever pro-

gramas, principalmente à medida em que eles crescem, tornando mais difícil assegurar que, em todas as partes do código, tudo está sendo feito corretamente.

**Este exemplo mostra essa instabilidade potencial:**

```
int main()
{
    struct Data pagamento ;
    struct Data vencimento ;

    pagamento = data_inicia ( pagamento ) ;
    data_imprime ( pagamento ) ; // imprimirá mensagem de erro: OK

    pagamento = data_altera ( pagamento , 1, 10, 2010 ) ;
    data_imprime ( pagamento ) ; // imprimirá a data ou erro: OK

    data_imprime ( vencimento ) ; // 🚬 o que será impresso aqui?
```



A variável '**vencimento**' está sendo impressa **sem que tenha sido iniciada ou alterada**.

A função '**data\_imprime**' irá avaliar o campo '**ok**', para saber se imprime a data ou uma mensagem de erro.

Mas qual será o valor desse campo? **Indeterminado** ("lixo").

Pois o seu valor atual não é resultado de qualquer **avaliação**.

```
// além disso, podemos:
// pagamento.dia = ... ; // atribuir livremente qualquer valor a um campo
// sem usar a função adequada.

return 0 ;
}
```



**Todos os problemas apontados acima são facilmente resolvidos em C++.**

### 7.1.3 • Passando estruturas como argumento

Os problemas apontados acima só serão adequadamente resolvidos em C++.

Mas há um outro problema que, mesmo em C, poderia ser resolvido de modo adequado: a maneira como a variável estruturada está sendo passada para as funções, pois essa é uma maneira ineficiente.

Para entender porque, vamos analisar qualquer uma das funções que estão recebendo uma cópia de uma variável do tipo "Data" como parâmetro.

Por exemplo, a função **data\_inicia**.

```
int main()
{
    struct Data dtHoje;
    dtHoje = data_inicia ( dtHoje );
    //...
}

struct Data data_inicia( struct Data dt )
{
    //...
    dt.ok = 0 ;
    return dt ;
}
```

Na função **main** criamos a variável **dtHoje** do tipo **struct Data**.

Em seguida chamamos a função **data\_inicia** passando a variável **dtHoje** como parâmetro.

O que ocorrerá aí será uma **cópia** do conteúdo armazenado na memória que apelidamos de “**dtHoje**” para uma outra memória, pertencente à função **data\_inicia**, a qual apelidamos de “**dt**”.

E **dt** é uma variável estruturada, com vários campos (e certamente essa estrutura teria ainda mais campos para atender a todas as funcionalidades necessárias).

Assim todos esses campos precisarão ser copiados de um lugar da memória para outro. E isto **reduz a performance**.

Além disso, a função **data\_inicia**, por sua vez, deve **retornar** uma **cópia** de **dt**, para que a função que a chamou possa receber as alterações que ela fez nesse parâmetro.

Por isso tivemos que fazer:

```
dtHoje = data_inicia ( dtHoje );
```

Pois, do contrário as alterações seriam feitas em **dt**, apelido de uma memória que pertence à função **data\_inicia** (e que é **liberada**, e com isso destruída, quando **data\_inicia** **retorna**).

Assim, para que o trabalho não seja perdido, deve haver uma nova cópia, desta vez em sentido inverso: desse modo o conteúdo de **dt** deve ser copiado para a área de retorno da função (o que é feito pela diretiva **return**) e, finalmente, deve ser copiado dessa área de retorno para a variável **dtHoje**, em **main** (o que é feito pelo operador de atribuição [=]).



Assim teremos **sucessivas cópias não de um pequeno valor** mas de uma variável estruturada que representa um **conjunto** de diversos valores (os seus campos), podendo ter um tamanho total de muitos **bytes**.

Por isso, devemos usar **outra forma de passagem de parâmetros** no caso de variáveis estruturadas: elas sempre terão um certo número de campos (muitas vezes, uma quantidade grande de campos). Seria bom que pudéssemos evitar a perda de tempo que ocorrerá com a cópia de todos os campos e também o desperdício da memória duplicada (com muitos **bytes**).



Esse objetivo pode ser atingido com **ponteiros (C e C++)** ou com **referências (C++)**.

Já vimos esse assunto (na seção **6.2.4**, página **195**).

**E vimos que há dois motivos para passar argumentos por endereço ou por referência:**

- Funções que devem retornar mais do que um valor: usam parâmetros "de saída" (ponteiros ou referências).
- Além disso, por motivos de **performance**: ao passar memórias com uma grande quantidade de **bytes**, devemos também definir o parâmetro da função como um **ponteiro** ou uma **referência**.

**Então as funções exemplificadas acima, deveriam ser escritas assim:**

```
void data_inicia( struct Data * pdt )
```

```

{
    // ...
    pdt->ok = 0 ;
}

void data_Altera ( struct Data * pdt, int dia, int mes, int ano ) ;
void data_imprime ( struct Data * pdt ) ;


```

### Ligação de membro a ponteiro para estrutura:

Para acessar os membros de dados de uma variável estruturada **a partir de um ponteiro** (como é feito acima com [ **pdt->ok = 0;** ]) é preciso usar o operador de ligação de membro corretamente.

**Assim, não poderíamos fazer:**

```

pdt.m_OK = FALSE ; //  ERRO: 'pdt' armazena um endereço,
                    // que é um número inteiro positivo.
                    // Logo, 'pdt' não possui campos...

```

Pois “pdt” é um **ponteiro**, e assim sendo ele simplesmente não tem campos... Logo, 'ok' **não é um membro** (ou um campo) de 'pdt'.

A variável 'pdt' está armazenando um endereço. E é **nesse endereço** que **estarão** os campos 'ok', 'dia', etc.

Logo **não** podemos ordenar: [ acesse o campo 'ok' de 'pdt' ].

- Pois a ordem **correta** é :
  - entenda '**pdt**' como o lugar onde está o endereço de uma variável do tipo '**struct Data**';
  - leia esse endereço;
  - agora vá até **esse** endereço, acessando esse **outro lugar** da memória;
  - e, agora sim, chegando **lá**, acesse o campo 'ok'.

Por isso, temos que usar o **operador ponteiro** para indicar que esse **acesso indireto** deve ser feito:

```
( * pdt ).ok = 0 ;
```

Nesses casos, como operações com ponteiros para estruturas são muito comuns, a linguagem fornece um operador específico (simbolizado por [ **->** ]).

Ele serve para ligar campos a ponteiros para estruturas. Não existe qualquer diferença de funcionamento. O operador específico é fornecido apenas para **simplificar a escrita** e torná-la mais legível.

**Assim, a melhor forma de uso é esta:**

```
pdt->ok = 0 ;
```

Como vimos também, **em C++**, **prefira referências a ponteiros**, sempre que isso seja possível.

Caso escrevêssemos uma função semelhante em C++, deveríamos fazer isto:

```

void data_inicia( Data & dt ) // referência
{
    // ...
    dt.ok = 0 ; // Se uma referência é formalmente um sinônimo
                // (e não um ponteiro), usamos normalmente o operador ponto

```



}

**Observe também que, em C++, não é necessário declarar a variável estruturada usando a palavra reservada 'struct' como parte do nome do tipo. Em C++ 'Data', simplesmente, já é considerada como o nome do tipo.**



E, como **regra geral**, estruturas **não** devem ser passadas por **valor**.  
E sim ou por **endereço**, ou, **preferencialmente (em C++)**, por **referência**.

### 7.1.4 • A especificação *const*

Resta um: quando passamos um endereço ou uma referência, a função que o recebe passa a ter acesso a uma variável declarada em outra função, **podendo alterá-la**

Se a função que recebe endereços ou referências tiver como objetivo justamente o fornecimento de retornos extras, então é exatamente isso que se deseja.

Mas se o objetivo for **apenas** velocidade de cópia(*performance*), então devemos deixar claro que essa função **não** irá alterar a memória cujo endereço ou referência ela recebeu e que usará esse endereço ou referência apenas para fins de **leitura** da variável apontada.

Assim poderemos ter certeza que, caso essa memória passe a apresentar comportamentos indevidos, a origem do problema poderá estar em qualquer lugar, menos na função que recebeu o endereço ou a referência exclusivamente para leitura.

Fazemos isso especificando ponteiros e referências em situações assim como **const**, o qual estabelece uma condição **read-only** para os dados

Nos exemplos acima, esse seria o caso da função **data\_imprime**.

Ela não foi projetada para alterar nada, e sim apenas para ler e imprimir.

Então, ela deveria ser declarada de uma das duas maneiras abaixo:

#### Usando ponteiros:

```
void data_imprime( const Data * pdt ) ; // Os dados apontados são
                                           // 'read-only' nesta função
```

#### Usando referências:

```
void data_imprime( const Data & dt ) ; // Os dados referenciados são
                                           // 'read-only' nesta função
```

## 7.2 • Estruturas em C++ : encapsulamento

Vimos que em C não há como garantir o encapsulamento de um modo simples. Já em C++ isso será possível.



Em **C++** podemos declarar tanto membros de dados(campos) como também as **próprias funções** dentro da declaração da **struct**, de modo que tanto os campos de dados como funções sejam membros da estrutura.

Além disso poderemos determinar que **apenas** funções declaradas dentro da **struct** possam acessar os membros de dados.

E, ainda, a **struct C++** permitirá uma **inicialização garantida** e segura que sempre **antecederá** qualquer tentativa de uso das variáveis estruturadas.


Essas características resolverão o problemas que já apontamos na estrutura **C**.  
E, além delas, teremos uma série de outros recursos – que resolverão outras insuficiências de **C**.

**Vejamos então como ficaria a nossa estrutura em C++.**

```
#include <string> // tudo aqui em um arquivo header (data.h, p.ex.)
                // já que poderá ser usada em muitos projetos.
```

```
struct Data
```

```
{
  private:
```


 Ao declarar membros com acesso **private**, estamos determinando que eles **só poderão ser acessados** nas **funções** declaradas **dentro desta struct** (ou, como veremos depois, em funções que **esta struct declare como "amigas"**, isto é, colaboradoras).

```
char    m_dia;
char    m_mes;
short   m_ano;

// variáveis de controle para validação do ano:
short m_anoMin , m_anoMax;


// flag que indica se a data está correta ou não:
bool    m_OK;      // C++ tem o tipo "bool" ...
```

```
public:
```


 Membros declarados como **public** podem ser acessados em **qualquer função**, esteja ela declarada aqui ou não (funções globais ou de outra classe).

```
// valores default para os anos mínimo e máximo, e constantes para meses:
enum { AnoMinDefault = 1 , AnoMaxDefault = 9999 } ;

enum { Janeiro=1, Fevereiro, Marco, Abril, Maio, Junho,
      Julho, Agosto, Setembro, Outubro, Novembro, Dezembro};
```

 A função abaixo ("**Data()**"), por ter **o mesmo nome da struct**, cumpre um papel **especial**: ela é classificada como **construtora** e será chamada automaticamente e obrigatoriamente sempre que uma variável deste tipo for criada (podendo assim ser usada para a **inicialização dos membros**, além da **aquisição de recursos** – como abertura de arquivos e outros)

```
Data ( /* parâmetro oculto */ ) ;
```

 **Sobrecarga: função com o mesmo nome, mas com tipos de parâmetros diferentes. Abaixo, temos uma *sobrecarga da construtora*. Podemos chamá-la de "construtora de conveniência" pois permite que os valores dia, mes e ano sejam passados ao criar (ou instancia) uma variável (objeto) do tipo "Data".**

```
Data ( /* parâmetro oculto , */ char dia, char mes, short ano ) ;
```

```
// outras funções-membro da struct
void altera ( /* parâmetro oculto , */
              char dia, char mes, short ano ) ;

char ultimoDiaMes (); // calcula o último dia do mês (m_mes)
bool anoBissexto (); // verifica se o ano (m_ano) é bissexto
std::string toString(); // formata a data em uma string
```

```
} ; // fim da definição da struct Data (note o ponto e vírgula após a chave)
```

### As funções membras da estrutura serão usadas assim:

```
int main()
{
    Data dtHoje ;    /* aqui é chamada automaticamente a função
                     "Data()" (construtora) , que receberá como
                     parâmetro oculto o endereço de "dtHoje". */

    dtHoje.altera( 1, 1, 2001) ; /* esta função receberá 4 argumentos:
                                3 passados explicitamente e mais um,
                                oculto, que é o endereço de "dtHoje". */

    Data dtOutra ( 2, 1, 2010 ) ; // aqui será chamada
                                // a "construtora de conveniência"

    // ...
}
```

 Anote:

- 1) Em C++ as **funções** fazem parte da estrutura, como **membros**
- 2) E o **acesso aos dados** deverá ser feito **obrigatoriamente** através de funções declaradas dentro da **struct** caso eles tenham sido declarados como privativos da estrutura (**private**).
- 3) No momento da declaração da variável (inicialização) é chamada automaticamente a função que tem o mesmo nome da estrutura (construtora).
- 4) Não precisamos (e não devemos) passar como parâmetro a variável estruturada ou o seu endereço: isto será feito **automaticamente** pelo compilador (pois o compilador acrescentará um **parâmetro oculto** contendo o **endereço** da variável estruturada que disparou a chamada à função).  
E o nome convencional para esse parâmetro oculto é "**this**"(que, em princípio, pode ser passado por registrador e não pela pilha).

Usamos as funções-membro com a mesma sintaxe com que usamos os membros de dados. Criamos uma variável usando o tipo estruturado em sua declaração. A partir daí simplesmente acessamos os membros de dados e funções, utilizando o **operador de ligação de membro**.

Só não devemos esquecer que dentro de funções não pertencentes à **struct** nunca poderemos acessar os membros declarados como **private** (sejam membros de dados ou funções-membro).

 Assim, no exemplo de "main", acima, temos a ressaltar que:



- Só podemos chamar a função "altera" a partir de uma **variável** já criada e que tenha sido **declarada** com o tipo "**Data**". O endereço dessa variável é então passado para a função.
- Mas essa chamada ocasionaria um erro de compilação se, na **struct**, "altera" fosse declarada como **private**.
- Assim, em uma função **externa** à **struct**, como é o caso de **main**, só podemos acessar os membros declarados como **public**.

## 7.3 • Implementando as funções membras

Agora, em um arquivo ".cpp" (data.cpp, p.ex.), para escrever a implementação das funções membras da **struct**, precisaremos indicar que elas **não são** funções globais ou


membras de uma outra **struct** qualquer, e sim, exatamente, funções membras **desta struct**.

Isso é feito através de um operador, o “operador de resolução de escopo”, que é representado pelo símbolo “::” (dois pontos duplos).

- 
-  Abaixo, a implementação da função **construtora**. Note que **construtoras não têm tipo de retorno** (são obrigatoriamente void e **não podemos** designar um tipo). Além disso, o **operador de resolução de escopo** [ :: ], usado abaixo indica: “função Data, **membra** da estrutura Data”
  -  E, ainda, antes do **corpo** da função ( { ... } ) devemos **inicializar os membros** que necessitem de inicialização obrigatória. Isso deve ser feito **iniciando com o operador de inicialização de membros (:) e usando vírgulas para os próximos membros**.
- 


*// os campos são acessados a partir do ponteiro “this”.\*//*

```
Data::Data( /* Data * this */ ) // parâmetro oculto cujo nome é “this”
    : m_anoMin(AnoMinDefault) // Inicialização de membros
    , m_anoMax(AnoMaxDefault)
    , m_OK(false) // C++ tem as palavras reservadas “true” e “false” ...
{
    // se necessário, adquirir recursos aqui (abrir arquivos, etc).
} // Nenhuma outra tarefa deve ser executada na construtora.

// Implementando a “construtora de conveniência”:
Data::Data( /* Data * this , */ char dia, char mes, short ano)
    : m_anoMin(AnoMinDefault) // Inicialização de membros
    , m_anoMax(AnoMaxDefault)
    // , m_OK(false) // não é preciso inicializar m_OK, pois abaixo
    //   será chamada a função “altera” que atribuirá um valor a m_OK
    //   de acordo com a correção ou não dos argumentos passados:
{
    altera ( dia, mes, ano ) ; 

    // se necessário, adquirir recursos aqui (abrir arquivos, etc).
} // Nenhuma outra tarefa deve ser executada na construtora.

// função “Altera”, da estrutura Data:
void Data::altera ( /* Data * this , */
                  char dia , char mes , short ano )
{
    m_dia  = dia ;
```

- 
-  Relembrando:  
O compilador C++ irá gerar o código necessário para a passagem do endereço da
-

variável que disparou a chamada a esta função - e que será recebido como **"this"** (parâmetro oculto).

E o compilador C++ **também** liga automaticamente os **campos** da struct ao **ponteiro** recebido.

Logo não precisamos fazer:

**this->m\_dia = dia;**

**Exceto** se o nome do membro fosse **"dia"** (o mesmo nome do parâmetro). **Nesse caso**, teríamos que fazer:

**this->dia = dia; // membro = argumento.** ☹

```
m_mes = mes;
m_ano = ano;

// análise e validação dos valores recebidos
// (obs: o exemplo usa 31, como último dia do mês. Mais a frente
// implementaremos o código para calcular o último dia de cada mês)

m_OK = m_ano >= m_anoMin && // and
       m_ano <= m_anoMax &&
       m_mes >= Janeiro && m_mes <= Dezembro &&
       m_dia >= 1 && m_dia <= 31 ; // depois, trocar 31
                                   // por uma chamada à função ultimoDiaMes
}
```

✂ **As demais funções declaradas na struct serão implementadas mais abaixo, após estudarmos o especificador const.**

## 7.4 • As palavras reservadas **struct** e **class**.

C++ tem duas palavras reservadas que fazem a mesma coisa, com uma pequena diferença: **struct** e **class**.

Podemos usar qualquer uma das duas para declarar uma estrutura de dados e funções.

A única diferença entre elas é que, na **struct** o acesso **"default"** é **public**, enquanto na **class** o acesso **"default"** é **private**.

✂ **Não existe qualquer outra diferença.**

*Assim, teríamos:*

```
struct Qualquer
{
    int x ; /* como não explicitamos o acesso, será usado o "default";
            logo, será public, já que usamos struct
            */
};

class Qualquer
{
    int x ; /* como não explicitamos o acesso, será usado o "default";
            logo, será private, já que usamos class
            */
};
```

Mais a frente veremos porque existem as duas palavras reservadas.

Por enquanto basta estabelecer que, em geral, é **melhor usar class** já que esta é mais segura (**se houver omissão**, vale o **private**, protegendo os membros contra acesso externo).

### 7.4.1 • Definindo o objeto apontado por “*this*” como *const*

Já vimos que se uma função recebe um ponteiro, mas precisa apenas **ler** a variável apontada, deixávamos isso claro especificando o ponteiro como **const**:

```
void Imprime ( const Data * pdt );
```

Mas, neste último caso, como fazer isso em uma **função-membro C++**, já que o endereço da variável é passado implicitamente através do parâmetro oculto **this**?

Para essa situação, a linguagem reservou uma sintaxe especial, já que o parâmetro é oculto:

```
<tipo> Nome_Funcao ( <parâmetros explícitos> ) const ;
```

O especificador **const** é colocado ao final do cabeçalho da função. E isso significa então que, nessa função, o **ponteiro *this* aponta para um objeto** recebido como **const (read-only)**.



**Observar que o ponteiro *this* é, por definição da linguagem, sempre *const*. Isto significa que não podemos alterar o endereço armazenado em *this*. Por exemplo:**

```
void Classe::func( )
```

```
{
```

```
    this = new Classe; //  ERRO: o this não pode ser alterado.
```

```
}
```

**O que significa que podemos entender essa função como:**

```
void Classe::func ( /* Classe * const this */ ) ;
```



**Mas se fizermos:**

```
void Classe::func( ) const;
```

**Neste caso além do ponteiro *this* ser inalterável, o objeto apontado por *this* também não pode ser alterado.**

**Podemos entender assim:**

```
void Classe::func ( /* const Classe * const this */ ) const;
```



**Em consequência teremos um erro aqui:**

```
void Classe::func ( /* const Classe * const this */ ) const
```

```
{
```

```
    // assumindo que “classeMembro” é um membro de “Classe”:
```

```

classeMembro = .... ; //  ERRO: o objeto apontado por this
// não pode ser alterado aqui.

}

```

Então, vamos aplicar o especificador **const** nas funções em que ele se aplique na **class Data**. Analisemos as funções:

- construtora: irá alterar dados, então **não** podemos usar **const**; e esta é uma regra geral para construtoras, já que seu papel é inicializar membros.
- "altera" : como o próprio nome está dizendo... **não** pode usar **const**.
- "ultimoDiaMes": não deve alterar dados, apenas irá apurar o último dia de cada mês; então deve ser especificada como **const**.
- "anoBissexto": idem, pois deverá apenas avaliar se o ano é bissexto;
- "toString": idem, pois apenas lê os campos para formatar uma string

## 7.4.2 • Implementação das funções membras **const**

Inicialmente, devemos alterar os protótipos das 3 últimas funções declaradas em Data ("data.h", por exemplo), e, também, trocar a palavra struct por class.

```

class Data
{
    .....

    // apurar o último dia de cada mês:

    char ultimoDiaMes( /* const Data * const this */ ) const ;
    // descobrir se um ano é bissexto:

    bool anoBissexto( /* const Data * const this */ ) const ;
    // converter dia, mês e ano para uma string e retornar esse string:
    std::string toString( /* const Data * const this */ ) const;
};

```

No arquivo **data.cpp** iremos escrever o corpo das funções:

```

#include <sstream> // streams que alimentam uma string
#include <iomanip> // manipuladores de IO
#include "data.h" // reconhecer os símbolos da class Data

// .....
// As funções construtoras e a função altera,
// já foram implementadas acima.

// Converter a data para uma string:

std::string Data::toString ( /* const Data * const this */ ) const
{
    std::ostringstream sout; // funciona como "cout" (classes de stream)
    // mas sua saída é um objeto-membro do tipo std::string

    if ( m_OK ) // se a data está correta (conforme o uso de "altera")
    {
        sout.fill('0'); // preenchedor à esquerda em função da largura(setw)
    }
}

```

```

sout << std::setw(2) << m_dia << '/'
    << std::setw(2) << m_mes << '/'
    << std::setw(4) << m_ano;
}
else
    sout << "***ERRO***";

```



Aqui talvez fosse necessário usar um **procedimento mais drástico**, pois este é um **erro sério**: tentando **recuperar dados** de um objeto que está em um **estado inválido**.



Veremos isso ao tratar do assunto **exceções**.

```

    return sout.str(); // retorna o membro std::string alimentado pelo stream
}

```

Abaixo, as demais funções, como exercício.



**Exercícios:**

- 1) Escreva a função “**anoBissexto**”, sabendo que:
- um ano é bissexto quando é divisível **por 4**
  - **mas não por 100, exceto se for divisível por 400.**

**Exemplos:**

2000 : **bissexto**, pois é divisível por 400;  
1800 : **não-bissexto**, pois não é divisível por 400, e, embora seja divisível por 4, **também** é divisível por 100.  
1996 : **bissexto**, divide por 4 mas não por 100  
1997 : **não-bissexto**, pois não é divisível por 4

- 2) Escreva a função “**ultimoDiaMes**”, sabendo que:
- fevereiro ..... : 28 ou 29 dias, se o ano for bissexto;
  - janeiro a julho ... : meses pares tem 30 dias, e ímpares 31 dias;
  - agosto a dezembro: meses pares tem 31 dias, e ímpares 30 dias;

**Exemplos:**

mês 6(junho)..... : anterior a agosto e par -> 30 dias;  
mês 7(julho) ..... : anterior a agosto e ímpar -> 31 dias;  
mês 8(agosto)..... : posterior a julho e par -> 31 dias;  
mês 9(setembro).... : posterior a julho e ímpar -> 30 dias;

Recursos que você já aprendeu e poderá usar:

- if ; else ;
- operador lógico “and” [ símbolo && ] ;
- operador de módulo (resto de divisão) [ símbolo % ]
- operador de bit’s “and” [ símbolo & ]
- operadores relacionais em geral( igual a, não-igual a, maior que, menor que ...)

♦ **Após finalizar os exercícios compare com as soluções expostas na próxima página.**

**Se tiver dúvidas anote e depois apresente-as ao instrutor.**

### *Solução para os exercícios.*

Serão apresentadas aqui diversas alternativas para cada um dos dois exercícios.

A idéia é que você perceba as diversas maneiras de usar alguns recursos básicos da linguagem. E assim **exercitaremos o uso dos operadores** e das condições **verdadeiro/falso** em C e C++.

Você pode escolher qualquer uma delas (ou então aquela que você desenvolveu) e usar no seu arquivo **data.cpp**. Contudo, há uma diferença de *performance* entre elas. Primeiramente é apresentada uma versão possivelmente menos eficiente (execução mais lenta) e, em seguida, uma possivelmente mais eficiente (execução mais rápida).

Assim, a última será sempre a mais eficiente:

a) anoBissexto:

```
// a.1) primeira versão:
bool Data::anoBissexto( ) const
{
    // se não for divisível por 4 não é bissexto:
    if ( m_ano % 4 != 0 ) // ou usando and de bits: if( (m_ano&3) !=0 )
        return false;

    // se divisível por 4 mas não por 100, exceto se divisível por 400, é bissexto:
    return m_ano % 100 != 0 || m_ano % 400 == 0;

    // nos dois casos acima não podemos usar o and de bits
    // pois, ao contrário de 4, os números 100 e 400 não são potências de 2
}

// a.2) segunda versão (usa o operador lógico OR [ símbolo: || ]):
bool Data::anoBissexto( ) const
{
    return m_ano % 400 == 0 || // OR
           ( m_ano % 4 == 0 && // AND
             m_ano % 100 != 0
           );
}

// a.3) terceira versão (usa o operador lógico NOT [ símbolo: ! ]):
bool Data::anoBissexto( ) const
{
    // o resto zero, nas divisões por 4 e por 400, diz que o ano é
    // (ou, no caso de 4, tem chances de ser) bissexto;
    // então inverte o zero (not zero)
    // para que o resultado seja verdadeiro (pois zero é falso)
    // [ !0 -> 1(true) ];
    // e na divisão por 100, o resto diferente de zero diz que o ano
    // tem chances de ser bissexto (então não é preciso fazer "!= 0") :
    return !(m_ano % 400) || // OR
           ( !(m_ano % 4) && // AND
             (m_ano % 100)
           );
}

/* a.4) quarta versão (procura executar as operações da mais frequente para
a menos frequente - afinal, a comparação com 400 colocada
em primeiro lugar, está privilegiando uma situação menos frequente):
*/

bool Data::anoBissexto( ) const
```

```

{
    return !(m_ano % 4) && // AND
           ( (m_ano % 100) || //OR
             !(m_ano % 400)
           );
}

/* a.5) quinta versão (usa o operador bitwise AND, para apurar
   o resto da divisão por 4, pois, nesse caso, através de um AND,
   bit a bit, entre o ano e a constante 3, é possível saber
   se é divisível por 4, sem usar o operador de módulo, mais lento,
   já que 4 é uma potência de 2.
*/

bool Data::anoBissexto( ) const
{
    /* Observe que:
    1 [0001] & 3 [0011] -> 0001 -> 1
    2 [0010] & 3 [0011] -> 0010 -> 2
    3 [0011] & 3 [0011] -> 0011 -> 3
    4 [0100] & 3 [0011] -> 0000 -> 0
    5 [0101] & 3 [0011] -> 0001 -> 1
    6 [0110] & 3 [0011] -> 0010 -> 2
    7 [0111] & 3 [0011] -> 0011 -> 3
    8 [1000] & 3 [0011] -> 0000 -> 0
    9 [1001] & 3 [0011] -> 0001 -> 1
    10 [1010] & 3 [0011] -> 0010 -> 2
    11 [1011] & 3 [0011] -> 0011 -> 3
    12 [1100] & 3 [0011] -> 0000 -> 0
    ... etc ...

    Enfim: apenas múltiplos de 4 em uma operação and, bit a bit,
    contra a constante 3, apresentam resultado zero.

    Então:
    */
    return !(m_ano & 3) && // AND
           ( (m_ano % 100) || //OR
             !(m_ano % 400)
           );
}

```

b) ultimoDiaMes:

*// b.1) primeira versão:*

```
char Data::ultimoDiaMes( ) const
{
    if ( m_mes == Fevereiro ) // 'Fevereiro' é uma constante da class
    {
        if ( anoBissexto() ) // se o ano for bissexto...
            return 29 ;
        return 28 ; // se chegou até aqui, é porque o ano não é bissexto.
    }

    // 'Janeiro' 'Julho', 'Agosto' e 'Dezembro' são constantes da class:
    // janeiro a julho, inclusive estes, exceto fevereiro
    if ( m_mes >= Janeiro && m_mes <= Julho )
    {
        if ( m_mes & 1 ) // se o mês é ímpar
            return 31 ;
        return 30 ; // se chegou até aqui, é porque o mês é par.
    }

    // agosto a dezembro, inclusive estes:
    if ( m_mes >= Agosto && m_mes <= Dezembro )
    {
        if ( m_mes & 1 ) // se o mês é ímpar
            return 30 ;
        return 31 ; // se chegou até aqui, é porque o mês é par
    }.

    return 0; // se chegou aqui, é um mês inválido
}
```

*/\* b.2) segunda versão:*

*(nesta versão usaremos o operador condicional ternário;  
esse operador funciona como um "if inline":  
(condicao)? resultado\_se\_condicao\_verdadeira  
: resultado\_se\_condicao\_falsa ;  
portanto o símbolo "?" é uma **pergunta** que exige resposta **verdadeira**  
e o símbolo ":"(dois pontos) significa "do contrário" (else).*

*\*/*

```
char Data::ultimoDiaMes( ) const
{
    if ( m_mes == Fevereiro )
        return ( anoBissexto() ) ? 29 : 28; // usa o operador condicional

    if ( m_mes >= Janeiro && <= Julho ) // janeiro a julho exceto fevereiro
        return ( m_mes & 1 ) ? 31 : 30;

    if ( m_mes >= Agosto && m_mes <= Dezembro )
        return ( m_mes & 1 ) ? 30 : 31;

    return 0; // mês inválido
}
```

```
// b.3) terceira versão (economiza testes de condição):
char Data::ultimoDiaMes( ) const
{
    if ( m_mes == Fevereiro )
        return 28 + anoBissexto(); // soma 1, se for bissexto, do contrário zero

    if ( m_mes >= Janeiro && m_mes <= Julho ) // exceto fevereiro
        return 30+( m_mes & 1 ); // soma 1, se for ímpar e zero se não for

    // agosto a dezembro:
    if ( m_mes >= Agosto && m_mes <= Dezembro )
        return 31-(m_mes & 1); // subtrai 1, se for ímpar e zero se não for.

    return 0; // mês inválido
}

// b.4) quarta versão (usa operador condicional no lugar do último 'if'):
char Data::ultimoDiaMes( ) const
{
    if ( m_mes < Janeiro || m_mes > Dezembro )
        return 0; // mês inválido

    if ( m_mes == Fevereiro )
        return 28 + anoBissexto();

    return ( m_mes <= Julho ) ? 30+ ( m_mes & 1 )
                               : 31 - ( m_mes & 1 );
}

/* b.5) quinta versão (usa o operador xor bit a bit).
```



Para entender esta versão consulte a explicação deste exemplo na seção sobre o operador *xor*, página **372**.

```
*/
char Data::ultimoDiaMes( ) const
{
    if ( m_mes < Janeiro || m_mes > Dezembro )
        return 0; // mês inválido

    return ( m_mes==FEVEREIRO ) ? 28 + anoBissexto( )
        : 30 + ( (m_mes & 1) ^ (m_mes > Julho) );

    // usando o xor bit a bit:
}
```

Após inserirmos, no arquivo **data.cpp**, uma das versões da “anoBissexto” e uma das versões da “ultimoDiaMes”(escolhidas entre as expostas acima, ou então use aquelas que você mesmo escreveu), teremos então concluído a primeira fase do nosso projeto “Data”.



**Não esqueça de trocar 31 por ultimoDiaMes() na função altera.**

*Já podemos testá-lo.*

### 7.4.3 • Testando a *class Data* (testadata.cpp)

O código para teste, a ser escrito no arquivo **testadata.cpp**, é o seguinte:

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

```

void imprime(const Data & dt ) // função auxiliar para imprimir Data
{
    std::cout << dt.toString() << '\n';
}

int main()
{
    Data d1;

    d1.altera(31,1,2001); imprime(d1); // resultado: 31/01/2001

    d1.altera(29,2,2001);    imprime(d1); // resultado: ERRO

    d1.altera(29,2,1997); imprime(d1); // resultado: ERRO

    d1.altera(29,2,1800); imprime(d1); // resultado: ERRO

    d1.altera(29,2,1996);    imprime(d1); // resultado: 29/02/1996

    d1.altera(29,2,2000); imprime(d1); // resultado: 29/02/2000

    d1.altera(31,6,2001); imprime(d1); // resultado: ERRO

    d1.altera(31,7,2001); imprime(d1); // resultado: 31/07/2001

    d1.altera(31,8,2001); imprime(d1); // resultado: 31/08/2001

    d1.altera(31,9,2001); imprime(d1); // resultado: ERRO

    d1.altera(31,12,2001); imprime(d1); // resultado: 31/12/2001

    return 0;
}

```

E o teste está correto, imprimindo as datas quando os dados estão bons e imprimindo mensagens de erro quando os dados estão inválidos.

#### 7.4.4 • Especificando funções como *inline*.

Quando uma função é especificada como "**inline**", isso significa que o compilador **podrá substituir** a sua chamada, diretamente pelo próprio **corpo** (implementação) da função. Já vimos isso acima (capítulo 5, página 131), com o exemplo da função **inline** "Minimo"

Na *class* "Data" podemos aplicar a especificação **inline** a algumas funções.

As duas candidatas são as funções "anoBissexto" e "ultimoDiaMes".

Pois, se estivermos usando as últimas versões propostas para elas mais acima (ou versões parecidas), teremos duas funções com apenas uma a duas linhas de código.

**Então poderiam ficar assim:**

```

inline bool anoBissexto( ) const ;
inline char ultimoDiaMes( ) const ;

```

**Lembrar que:**



A especificação **inline** solicita ao compilador que no lugar da chamada da função, ele insira, diretamente, o próprio código da função. Assim, funções **inline** devem ser implementadas nos arquivos **header (.h)**, ou em arquivos por ele incluídos, de modo que seu corpo esteja **disponível** para o compilador **caso ele opte** por efetuar a substituição.

**Em consequência**, deve ser feito o seguinte (no arquivo **data.h**):

```
class Data
{
    .....
    // protótipos das funções inline:
    inline bool anoBissexto( ) const ;
    inline char ultimoDiaMes( ) const ;
};

// implementação das funções inline que serão usadas em outros módulos,
// deve ser feita aqui, no próprio arquivo data.h, após a declaração da classe,
// ou então em um outro arquivo aqui incluído:

inline bool Data::anoBissexto( ) const
{
    .....
}

inline char Data::ultimoDiaMes( ) const
{
    .....
}
```

### 7.4.5 • Usando um segundo parâmetro do tipo “Data”.

Na estrutura C++ as funções-membro recebem como primeiro parâmetro oculto o endereço da variável a partir da qual a função foi chamada. É o ponteiro *this*.

Em algumas funções precisaremos também de uma segunda variável desse tipo estruturado.

Por exemplo, se precisássemos comparar duas “Datas”, para saber qual é a mais antiga(menor) ou qual é a mais atual(maior), poderíamos ter a seguinte função membro:

```
int Compara( /* const Data * this , */ const Data * other) const;

/* retorna :      0   (zero) se estiverem iguais;
                  < 0  se a primeira estiver menor;
                  > 0  se a primeira estiver maior.
*/
```

**Que seria usada assim:**

```
Data dtPagamento ;
Data dtVencimento ;

if ( dtPagamento.Compara( &dtVencimento ) == 0 )
    printf ( "Datas iguais\n");
```

Nesse caso, o endereço de **dtPagamento** é passado no primeiro parâmetro (o parâmetro oculto **this**), e o endereço de **dtVencimento** é passado explicitamente como segundo parâmetro (**pdt2**).

Mas já sabemos que, em C++, há uma forma mais eficiente de fazer isso. Podemos (e devemos) passar o segundo parâmetro por **referência** (ao invés de endereço),

Então, Na declaração da classe (**data.h**), acrescente o seguinte protótipo, onde o segundo argumento é uma **referência**:

```
// na função prototipada abaixo,
// o parâmetro 'this' é um ponteiro
// que aponta para um objeto especificado como const
// e o parâmetro 'other' é uma referência (especificada como const):
int Compara( const Data & other ) const;

/* retorna :      0  (zero) se estiverem iguais;
                  < 0  se a primeira estiver menor;
                  > 0  se a primeira estiver maior.
*/
```

### Exercício:

- 1 - No arquivo **data.cpp**, escreva a função “**Compara**”.
- 2 - Depois, no arquivo **testadata.cpp**, acrescente algumas linhas para testar a nova função.

Nesta versão, ainda não temos um número único, que simplificaria a comparação.

Então será preciso comparar ano, mês e dia separadamente, sabendo-se que:

- ano vale mais que mês;
- mês vale mais que dia.
- **retorno** da função : leia o comentário acima, logo abaixo do protótipo.

♦ **Depois compare a sua solução com as duas soluções que são expostas na próxima página.**



**Função “Compara” (arquivo data.cpp):**

```

// a) versão 1:
int Data::Compara( /* const Data * this , */    const Data & other ) const
{
    if ( this->m_ano > other.m_ano )
        return 1; // primeira maior
    // o this é assumido por default... Então:
    if ( m_ano < other.m_ano )
        return -1; // primeira menor
    if ( m_mes > other.m_mes )
        return 1; // primeira maior
    if ( m_mes < other.m_mes )
        return -1; // primeira menor
    if ( m_dia > other.m_dia )
        return 1; // primeira maior
    if ( m_dia < other.m_dia )
        return -1; // primeira menor
    return 0; // iguais
}

// b) versão 2:
int Data::Compara( /* const Data * this , */    const Data & other ) const
{
    if ( this->m_ano != other.m_ano )
        return this->m_ano - other.m_ano ;
    return ( this->m_mes != other.m_mes )
        ? this->m_mes - other.m_mes
        : this->m_dia - other.m_dia ;

    // e o retorno será:
    // menor que zero se a primeira estiver menor
    // maior que zero se a primeira estiver maior
    // zero, se estiverem iguais.
}

```

No arquivo **testadata.cpp**, podemos testar a nova função.

No topo do arquivo, acrescente:

```
#include <iostream> // std::cout será usada em main
```

E, antes do **return** de **main**, acrescente, por exemplo:

```

Data d2;
d2.altera(1, 1, 2002 );
int comp = d1.Compara(d2);

if ( comp > 0 )
    std::cout << "d1 maior\n";
else if ( comp < 0 )
    std::cout << "d1 menor\n";
else
    std::cout << "iguais\n";

```

**7.4.6 • Acrescentando funções operadoras à classe “Data”.**

A estrutura C++ permite a escrita de funções no formato operador.

Isto torna o uso de algumas funções **mais legível** (leitura de código mais rápida) e também permitirá que **classes diferentes se comuniquem** (pois operadores são uma linguagem universal).

Assim podemos escrever, dentro da declaração da **class Data**, no arquivo **data.h**:

```
bool operator == (const Data & other ) const;
```

Que poderia ser usado de duas maneiras:

```
Data d1, d2;
// 1) primeira forma de usar
// (evite esta forma, exceto dentro da própria "class"):
if ( d1.operator==(d2) )
    std::cout << "iguais\n" ;

// 2) segunda forma de usar (e esta é a forma comum de uso):
if ( d1 == d2 ) // muito mais legível (e também mais legível
                // que "Compara")
    std::cout << "iguais\n";
```

#### Note que:

- o endereço do **primeiro** operando(**d1**, no exemplo acima) será passado como **this**;
- o **segundo** operando (**d2**, no exemplo acima), será passado como parâmetro **explícito** (e por referência no exemplo acima).

Implementação da função (arquivo **data.cpp**):

```
bool Data::operator == ( const Data & other ) const
{
    return ( this->Compara( other ) == 0 );
}

// OU, simplesmente:
bool Data::operator == ( const Data & other ) const
{
    return ( Compara ( other ) == 0 );
}
```



A função operadora acima, apenas chama uma outra função. Assim, deve ser declarada como **inline** para **evitar** uma **dupla chamada** de função,

// *protótipo, dentro da declaração da classe (arquivo data.h):*

```
class Data
{
    .....
    ➡ inline bool operator == ( const Data & other ) const ;
    .....
};
```

// *implementação, no próprio arquivo "header" data.h, **após** a definição da classe ou em outro arquivo, aqui incluído:*

```
➡ inline bool Data::operator == ( const Data & other ) const
{
    return ( Compara ( other ) == 0 );
}
```

**Exercício:**

- 1 - Declare (em **data.h**) e implemente(em **data.cpp**) os demais operadores relacionais (*não-igual ; maior que ; menor que ; maior ou igual que ; menor ou igual que*).
- 2 - Depois, no arquivo **testadata.cpp**, acrescente algumas linhas para testar os novos operadores.

*Exemplo (operador menor que):*

```
inline bool Data::operator < ( const Data & other ) const
{
    return ( Compara ( other ) < 0 );
}
```

### 7.4.7 • Os operadores << e >> de ostream/istream

Agora já podemos entender melhor porque temos feito:

```
std::cout << "saída\n";
int x; std::cin >> x ;
```

Isto é possível porque, como vimos, podemos criar operadores para um determinado tipo de dados (uma **struct** ou **class**).

- A biblioteca padrão do **C++** oferece, ao invés de funções globais (como é o caso da biblioteca padrão do **C**), **estruturas de dados e funções** para nos fornecer suporte em uma série de situações básicas, entre elas as operações de entrada e saída.
- Assim, podemos usar uma **classe C++** para **entrada** de dados (**class istream**) e uma outra para **saída** (**class ostream**).
- E, ainda, a biblioteca padrão declara duas **variáveis globais** (que podem ser usadas em qualquer ponto da aplicação) usando essas duas classes para tipificá-las:

```
ostream cout;
istream cin;
```

- Logo, a partir dessas variáveis já criadas, podemos usar suas **funções-membro e operadores**.
- E, nessas classes de entrada e saída, estão definidos **operadores de direção de fluxo** (**shift** ou deslocamento), que funcionam como um inseritor no fluxo.

Assim:

```
std::cout << "Vai para a saída primeiro" << "Vai para a saída depois"
<< std::endl ;
```

- **cout envia** para a saída padrão (o monitor de vídeo) as duas cadeias de caracteres que ele **recebeu** seguindo a ordem sequencial de envio (como em um fluxo).
- Finalmente é enviado um terminador de linha e um "flush": é o **"endl"**;

*Outro exemplo:*

```
int iImprimePrimeiro , iImprimeDepois;
std::cout << iImprimePrimeiro << iImprimeDepois << std::endl ;
```

- E o mesmo ocorre para entrada de dados (invertendo-se a direção do fluxo):

```
int iEntraPrimeiro, iEntraDepois ;
std::cin >> iEntraPrimeiro >> iEntraDepois ;
```

- **cin** receberá dois valores da entrada padrão (o teclado) e em seguida os **enviará** para armazenamento nas variáveis **iEntraPrimeiro** e **iEntraDepois**, segundo a **ordem** de entrada.

### 7.4.8 • Conclusão Parcial

A linguagem C++ permite que criemos nossos próprios tipo de dados a partir de tipos já existentes. Inicialmente os tipos existentes são os tipos primários oferecidos pela linguagem e, como logo veremos, também os tipos e gabaritos de tipos da biblioteca padrão C++.

Contudo à medida que criamos tipos novos, eles também poderão ser usados na criação de outros tipos.

Um tipo novo é criado através das palavras reservadas **enum** ou **struct**. Há também a palavra reservada **union** que permite criar tipos, mas cujo uso em C++ é menos frequente do que em C.

A diretiva **enum** cumpre um papel específico, podendo apenas criar um tipo numérico associado a uma lista de símbolos para constantes.

Já a diretiva **struct** cumpre um papel muito mais amplo.

Pois, através dela, podemos criar conjuntos de dados de quaisquer tipos existentes.

E a **struct** irá também unir os dados ao código, permitindo que os dados sejam protegidos pelo código, principalmente através das restrições de acesso **private** e **public** – conforme vimos na primeira parte da apostila.

Assim, é através de **struct** que poderemos implementar os principais conceitos de orientação a objeto em C++: encapsulamento, herança e polimorfismo.

Em resumo: em C++ os conceitos da orientação a objetos são implementados principalmente através de **tipos de dados**.

Mas, além de orientação a objetos, C++ oferece também **recursos para programação genérica**

- **1 - “gabaritos” para tipos (templates)**, que permitem o desenvolvimento de algoritmos genéricos sem preocupação com o tipo (quando um algoritmo se aplica a diversos tipos), mas que, depois, só poderão ser usados a partir de tipos claramente especificados.
- **2 – A STL (Standard Template Library)**, baseada em **templates**, e que já fornece, prontos, recursos importantes para programação genérica, os quais devem ser ampliados com o novo padrão (C++0x)

E programação genérica também pode ser combinada com orientação a objetos, pois podemos ter *templates de classes*.



Além disso, como veremos, **operadores** não são bons apenas por legibilidade. Mas principalmente porque, dão um importante suporte à programação genérica, quando usados por *templates*.



Inicialmente vamos recapitular o básico: classes em C++ fornecem novos *tipos de dados*. Vejamos mais detidamente os conceitos aí envolvidos.

## 7.5 • Encapsulamento: reforçando conceitos.

Vamos aqui recapitular os recursos da linguagem que permitem implementar a técnica de **encapsulamento** (proteção de dados e ocultamento de recursos que não devem ser usados fora da classe).

Em seguida iremos ver alguns recursos adicionais.

Recapitulando. Os recursos de encapsulamento vistos até aqui são:

- struct e class
- restrições de acesso
- membros de dados
- funções membras
- função construtora

Vejamos agora, com mais detalhes, alguns tópicos.

### 7.5.1 • Funções inline.



Já vimos funções *inline*. Mas, falando em encapsulamento, é importante ressaltar o papel cumprido por elas para que o encapsulamento não leve a perda de eficiência.

Aparentemente, quando declaramos um membro de dados com restrição de acesso **private**, passamos a ter um problema de *performance*, pois em muitos casos **teremos funções cujo único objetivo é retornar uma cópia dos membros de dados declarados como private**.

Estamos desperdiçando processamento (chamando uma função e retornando dela) para manter o encapsulamento.

Mas isso pode ser resolvido se a função for declarada como **inline**.

Nesse sentido, ela auxilia o encapsulamento, evitando a tentação de declarar membros de dados como **public** para evitar perdas de processamento com chamadas de função.

### 7.5.2 • O ponteiro this.



Como realmente funciona a chamada a uma função-membro?

- Quando escrevemos (supondo-se uma classe "Menu", que contenha uma função "exibe"):
  - Menu novoMenu;
  - novoMenu.exibe( )
- O compilador traduz (aproximadamente) para:
  - Menu::exibe( & novoMenu);
  - no código de máquina, isso ficaria **aproximadamente** assim (mas isso depende de compilador e convenções adicionais):
    - passa o **endereço de 'novoMenu'**, e em seguida chama a função:  
**call exibe@Menu**

---

✎ Ou seja: o nome da **class** fará parte do nome completo da função (também uma codificação dos tipos de seus parâmetros, mas isso não está em foco aqui).

---

□ O que significa:

---

✎ "Chamada à função exibe, da Classe Menu, passando como parâmetro o endereço do objeto que acionou a função".

---

- Então, significa que o **endereço do objeto** é passado como um parâmetro **implícito e automático** para a função membro.
- E, seja qual for a classe, em qualquer função-membro de classe podemos ter acesso a esse parâmetro.
  - Ele é representado pelo ponteiro **this**;
  - É um ponteiro fixo: não podemos modificar o endereço que ele armazena;
  - Mas, sempre que for necessário acessar o objeto que acionou a função (por exemplo, para repassá-lo para uma outra função qualquer) poderemos lançar mão do ponteiro **this**.

### 7.5.2.1 ▪ Definindo o objeto apontado por **this** como **const**

Quando escrevemos uma função que recebe um **ponteiro** como **parâmetro**, mas o **valor por ele apontado não** será alterado pela função, deixamos claro esse fato especificando o parâmetro como **const**.

E como fazer isso com o parâmetro **this**, já que ele é **oculto**?

Fazemos isso, acrescentando o especificador **const no fim do cabeçalho da função. Essa é a sintaxe determinada por C++ para tal finalidade.**

**Exemplo:**

```
class Data
{
    .....
    std::string toString( ) const ;
    .....
}
```

No exemplo acima, como o objeto apontado pelo ponteiro **this** foi especificado como **const**, os valores por ele apontado **não poderão ser modificados**.

E como o **this** é o endereço de uma variável estruturada que será usada para chamar a função, isso significa que os campos dessa variável **não** serão alterados pela função "*toString*".

```
.....
Data qualquer (1, 1, 2012 ) ;
std::cout << qualquer.toString() << '\n' ;
.....
```

O endereço da variável *qualquer* será passado como primeiro parâmetro (**this**) para a função *toString* da classe *Data*.

E, na função *toString*, o objeto apontado pelo ponteiro **this** foi especificado como **const**; assim é garantido que os membros de dados da variável *qualquer* **não** serão alterados por essa função.

### 7.5.2.2 • Exceção para a restrição **const** de *this* (*mutable*)


Podemos desejar que **alguns** membros possam ser alterados mesmo em funções membro em que objeto apontado pelo ponteiro **this** é especificado como **const**.

Estes membros devem ser declarados usando-se a palavra reservada **mutable**.

#### *Exemplo:*

```
class Qualquer
{
    .....
    int var;
    mutable int podeAlterarSempre;
    .....

    void Imprime( ) const
    {
        var=0; // ERRO: o objeto apontado por this é const e var não é mutable.
        podeAlterarSempre = 0; // OK: este membro é mutable.
        .....
    }
};
```

 Essa é uma especificação arbitrária. Mas que, em alguns casos pode ser muito útil (geralmente, em situações de herança, como veremos).

### 7.5.3 • Função Construtora

Considerando o seguinte exemplo:

```
class Menu
{
    .....
    Menu();
    .....
};
```

- Nesse exemplo da classe *Menu*, podemos observar que há uma função que tem **o mesmo nome da classe**: a função **Menu( )**.
- Por ter o mesmo nome da classe, esta função é chamada **construtora** e tem um significado especial:
- **Ela só pode ser chamada no momento da criação de uma variável**;
- Assim, se escrevermos:
 

```
Menu NovoMenu ;
NovoMenu.Menu( ) ;
```
- O resultado será um **erro de compilação**.
- Isso porque uma construtora é chamada **automaticamente na declaração** da variável, isto é, no momento em que a variável é criada na memória.
- **Desse modo** temos as seguintes possibilidades:

**Menu NovoMenu ;** // a função construtora é chamada **aqui**

```

Menu * pNovoMenu = new Menu; // a função construtora
                             é chamada aqui

Menu( ); // é criado um objeto temporário (sem nome)
         // sendo executada a construtora;
         // em seguida o objeto é destruído

// Na linha abaixo a construtora será chamada também para a criação
// de um objeto temporário(anônimo) que será copiado para o
// parâmetro da função "Gravar":
Gravar ( Menu( ) );
/*
1) a função Gravar recebe um parâmetro do tipo Menu
2) quando chamamos a função Gravar, passamos como parâmetro
   um objeto menu criado temporariamente (sem nome), o que provoca uma
   chamada à construtora.
*/

```



Observe que nos **quatro exemplos acima, o princípio é o mesmo**: a construtora é chamada **no momento da criação da variável e apenas nesse momento**.

- Como as construtoras são automaticamente chamadas quando um objeto (uma variável desse tipo) é alocada na memória, a consequência é que as construtoras serão especialmente úteis para inicializar os membros de dados daquele objeto que está sendo criado, seja com simples valores default, seja a partir de parâmetros.
- Pois uma declaração de variável implica em que naquele ponto do processamento será reservada a memória necessária determinada pelo tipo da variável. No caso de uma classe, isto significa que será usado o seu molde para alocar memória para todos os **membros de dados**.
- **E, em seguida, imediatamente após a alocação da memória** será sempre chamada a função construtora.
- Desse modo, a construtora é chamada uma **única vez** para cada objeto que é criado.
- Como é chamada na linha de declaração (criação), a chamada é **visível no código fonte**; assim podemos passar **parâmetros** para uma construtora.

### 7.5.3.1 • Parâmetros para a função construtora.

Uma função construtora pode receber parâmetros, normalmente usados para inicializar os membros de dados.

Em C++ podemos ter diversas funções com o mesmo nome **desde que sua parametragem seja diferente**, pois tanto o nome da **class** quanto os **tipos** dos parâmetros **são usados** na composição do nome real da função em código de máquina.

Assim podemos ter diversas “**versões**” da construtora (na realidade diversas funções construtoras com o mesmo nome mas com quantidade e/ou tipos de parâmetros diferentes).

### 7.5.3.2 • Construtora *default*.

A construtora **sem parâmetros** é considerada a construtora *default*.

Isto significa que em determinadas situações poderemos ter uma chamada à construtora não escrita diretamente no código fonte (será chamada a construtora *default*).

Entenderemos melhor a utilidade da construtora *default* quando estudarmos **herança**.



Mesmo que não tenhamos escrito nenhuma construtora, a construtora *default* é implementada pelo compilador (caso necessário, isto é, caso seja usada).

Contudo, escrevermos **alguma** construtora com parâmetros, o compilador não implementará automaticamente a construtora *default*.


Neste caso, precisaremos implementar também uma construtora *default*.

### 7.5.3.3 • Construtora de cópia e operador de atribuição.

Além da construtora *default*, uma outra construtora é implementada pelo compilador automaticamente: a construtora **de cópia**.

Esta construtora permite que criemos um objeto a partir do outro **do mesmo tipo**.

```
class Qualquer
{
    // nenhuma construtora foi declarada
    .....
};
.....
Qualquer q1;
Qualquer q2 ( q1 );
```

 O objeto "q2" será construído e receberá uma cópia, campo a campo, do objeto "q1".

O mesmo ocorre com o **operador de atribuição**. Ele é implementado automaticamente pelo compilador para permitir **cópias dos campos de um objeto para os campos correspondentes de um outro objeto do mesmo tipo**.

**Exemplo:**

```
.....
Qualquer q1;
Qualquer q2;
q1 = q2;
.....
// os campos de q2 serão copiados para
// os campos correspondentes de q1
```

Na realidade, a implementação do construtor de cópia e do operador de atribuição para cópia é muito simples. É feita uma cópia byte a byte, do endereço inicial de um objeto para o endereço inicial do outro (geralmente usando a função *memcpy*), usando-se aí o tamanho da estrutura (*sizeof*), para determinar quantos bytes devem ser copiados.

Caso precisemos de um operador de atribuição que receba como argumento um **tipo diferente**, então deveremos declará-lo e implementá-lo explicitamente.



Mas cuidado: se algum membro de dados for responsável pela alocação de *recursos adquiridos manualmente* (arquivo, memória dinâmica, etc) a cópia estará acessando esse recurso (tanto devido ao construtor de cópia como ao operador de atribuição).

Pois esse recurso pode ser destruído a qualquer momento por um dos objetos (o original ou a cópia) sem que o outro objeto fique sabendo disso.



Se existirem recursos assim em uma classe temos duas soluções:

- Ou implementamos o construtor de cópia e o operador de atribuição explicitamente, de um modo tal que resolva o problema e que **que faça sentido**,
- ou, caso isso não seja possível, **devemos anular o construtor de cópia e o operador de atribuição** declarando-os explicitamente como membros **private e sem implementação (apenas o protótipo)**.

*Exemplo:*

```
class Qualquer()
{
    //...
    private
        // Construtor de cópia: private e apenas prototipado:
        // não terá implementação:
        Qualquer( const Qualquer & );
        // Idem para o operador de atribuição:
        Qualquer & operator = ( const Qualquer & );
};
```

#### 7.5.3.4 • Construtoras com conversão implícita e explícita.

Vejam os a seguinte implementação da classe **Qualquer**:

```
class Qualquer
{
    private:
        int x;

    public:
        Qualquer ( int i ) // construtora recebendo um parâmetro
            : x ( i )
        {
        }
    .....
};
```

Agora, temos uma função que recebe como parâmetro um objeto do tipo **Qualquer**:

```
void Funcao( Qualquer qq )
{
    .....
}
```

E chamamos essa função passando como parâmetro um valor **do tipo int**:

```
.....
Funcao ( 5 );
```

Será criado um objeto temporário **do tipo Qualquer**, usando a construtora que recebe um **int** como parâmetro.

O objeto temporário será em seguida copiado para o parâmetro “**qq**” de “**Funcao**”.

Em alguns casos esse comportamento é indesejável, podendo criar situações fora de controle. Em outros casos essa escrita estaria indicando um erro de compilação (a classe não foi projetada para ser usada **desse modo**).

Para prevenir esse comportamento automático de conversão implícita, podemos proibir as conversões implícitas, declarando a construtora com a palavra reservada **explicit**.

```

class Qualquer
{
    private:
        int x;

    public:
        explicit Qualquer ( int i ) // a construtora só admite
                                   // conversõesEXPLÍCITAS.

            : x ( i )
        {
        }
        .....
};

```

**Agora se tentarmos fazer:**

```

void Funcao( Qualquer qq )
{
    .....
}
.....
Funcao ( 5 );

```

Obteremos um **erro de compilação**.

Para corrigir o erro teremos que ser **explícitos**:

```

Funcao ( Qualquer( 5 ) );

OU:

Qualquer qq ( 1 ); Funcao ( qq );
// OK: nos dois casos, estamos explicitamente construindo
// um objeto do tipo Qualquer
// e passando um inteiro como argumento para a construtora.

```

### 7.5.3.5 • Delegando construtoras (C++11)

Uma construtora pode delegar a inicialização de outra construtora, ou seja, ao uma construtora ser chamada pode disparar outra, só não pode delegar ela mesma e não pode conter mais nada na lista de inicialização. Uma construtora pode delegar outra que delega outra e assim por diante. O corpo da construtora que delegou será executado assim que a construtora delegada terminar.

```

#include <iostream>
using namespace std;

class Teste
{
    int a;
    int b;
public:
    // Delegando (chamando outra construtora
    // OBS: Nao pode conter nenhuma outra inicialização a
    // não ser de construtoras
    Teste() : Teste(0)

```

```

{
    cout << "Sem parametro\n\n";
}
Teste(int a) : Teste(a, 0)
{
    cout << "Com 1 parametro\n\n";
}
Teste(int a, int b) : a(a), b(b)
{
    cout << "Com 2 parametros\n\n";
}
};

int main()
{
    Teste t;
    return 0;
}

```

### 7.5.4 • Função destrutora

- Uma função destrutora também tem o mesmo nome da classe precedido pelo sinal `~(til)`;
- A função destrutora é chamada automaticamente quando a variável (o objeto) vai ser destruída:
  - ou porque a variável vai **sair da abrangência** (fim do bloco de sua declaração);
  - ou porque foi criada através de **new** e agora foi empregado um **delete**.
- Como ela é chamada automaticamente no momento da destruição, sua chamada é invisível no código fonte; logo **não pode receber parâmetros (e, por isso mesmo, não podemos ter mais do que uma destrutora)**.
- Devemos usar destrutoras para qualquer finalização necessária.



Em especial para liberar recursos adquiridos manualmente, como memória dinâmica, arquivos, etc.

- A destrutora também é automaticamente chamada quando usamos **delete** tendo como argumento um ponteiro para um objeto alocado no **heap**.

```

int main()
{
    .....
    Menu * pNovoMenu = new Menu;
    .....
    delete pNovoMenu;    // a memória apontada por pNovoMenu
                        // será liberada aqui e a destrutora será chamada
    .....
    return 0;
}

```



Podemos também chamar uma destrutora **diretamente**. Mas isso significa que irão ocorrer duas chamadas à destrutora (pois ela sempre será chamada automaticamente quando a memória for liberada).



A chamada direta à destrutora se justifica em determinados casos, geralmente em algoritmos complexos, onde a chamada direta simplifica a lógica (por exemplo, alguns algoritmos da biblioteca padrão usam chamadas diretas à destrutora).

## 7.5.5 ▪ Alterando o ponto de entrada da aplicação: objetos externos.

Como vimos acima, construtoras e destrutora funcionam exatamente como **respostas a eventos** (respectivamente, a criação e a destruição de um objeto).

O compilador insere no ponto em que um objeto é criado uma chamada à construtora. E, no local em que será liberado, insere uma chamada a sua destrutora.

Assim, uma construtora é chamada no momento em que um objeto é criado. Vejamos como isso funciona com variáveis **externas**.



Variáveis **externas** (globais ou modulares) são alocadas na memória **antes** de qualquer outra coisa – antes mesmo que a função **main** seja chamada.



Assim sendo, quando declaramos uma variável externa de um tipo estruturado será chamada a sua construtora no momento de sua criação, o que significa que ela será executada **antes de main**.

Desse modo o **ponto de entrada da aplicação**, embora **formalmente** pertença a **main** (já que uma chamada a **main** é sempre incluída pelo compilador), na prática será transferido para a **construtora do primeiro objeto externo** que tenha sido declarado (o que pode ser difícil de determinar se houver muitos arquivos fonte que declarem objetos globais)..

E uma variável externa só é liberada **após o retorno** de **main**. Isso significa que **sua destrutora só será chamada após esse retorno**.

Desse modo o último código escrito pelo programador a ser executado será a **destrutora do primeiro objeto externo** que tenha sido declarado.

Assim sendo, concluímos que um **objeto tem sua própria lógica**, alterando o fluxo de processamento convencional.



Dependendo do caso, isso pode levar a **erros**. Por exemplo, se "main" liberar recursos que são requeridos pelo objeto global.

Como regra geral, devemos evitar. Caso necessário, por alguma razão, seria melhor ter um ponteiro global para o objeto, mas que só seria instanciado no **início de main**, sendo liberado ao **final de main**.

## 7.5.6 ▪ Classes e funções amigas

Quando uma classe declara uma outra classe, ou uma função externa, como amiga, está permitindo que o declarado tenha acesso direto a todos os membros da classe que declarou, inclusive os membros private.

```
class X
{
```

```
friend int main( ) ; //a função main é uma função amiga
friend class Y ;    // a classe Y é uma classe amiga;
};
```



No exemplo acima, tanto a *função **main*** quanto as *funções-membras da classe **Y*** podem acessar os dados e funções **private** da classe **X**.



É preciso portanto cuidado ao usar declarações **friend**.

Uma má utilização irá eliminar o encapsulamento da classe que declara, abrindo caminho para efeitos colaterais.

Uma classe ou função amiga, faz sentido quando precisamos de um recurso auxiliar a uma determinada classe.

O recurso auxiliar é **conhecido pela classe** e é seu ajudante.

Um exemplo típico seria o seguinte:

```
class Node
{
    friend class NodeList ; // declaração friend
    //...
}

class NodeList // uma lista de "Nodes"
{
    Node * m_node;
    //...
}
```

Observar que nesse caso precisamos de duas classes para resolver o nosso problema.

Pois “NodeList” precisa de um **conjunto de “Nodes”**. Então a classe *Node* só existe para auxiliar *NodeList*.

Faz sentido que ela declare *NodeList* como *friend*.

## 7.5.7 ▪ Membros estáticos de uma classe

### 7.5.7.1 ▪ Membros de dados estáticos



Um membro de dados estático representa uma **memória compartilhada por todos os objetos da classe**. Do ponto de vista do **tempo de vida** tem o mesmo comportamento de uma variável global.

Contudo, o membro estático **obedece ao escopo da class** (seu nome) e está sujeito às **restrições de acesso** (private).

Por isso **prefira** membros estáticos a objetos globais.

*assim se fizermos:*

```
class Qualquer
{
    .....
    public:
        static int MembroStat ;
    .....
};
```

*e depois:*

**int Qualquer::MembroStat = 0; // inicialização no escopo global**

```
int main( )
{
    // sendo estático pode ser acessado mesmo que nenhum objeto exista.
    std::cout << Qualquer::MembroStat << '\n'; // imprime 0

    Qualquer Coisa ;
    Qualquer Outra ;

    Coisa.MembroStat = 5;
    cout << Coisa.MembroStat << endl; // o resultado é 5
    cout << Outra.MembroStat << endl; // o resultado é 5

    Outra.MembroStat = 10;
    cout << Coisa.MembroStat << endl; // o resultado é 10
    cout << Outra.MembroStat << endl; // o resultado é 10
}
```



verificamos que tanto faz o objeto usado para acessar o membro: cada alteração feita nesse membro estático em qualquer objeto afeta todos os demais objetos dessa classe.



O membro estático é o **único** para todos os objetos de sua classe.

Justamente por isso um membro estático deve, **obrigatoriamente**, ser inicializado **fora da classe (não na construtora)**.



Se fosse inicializado em uma função construtora, estaríamos sempre, a cada novo objeto criado, alterando seu valor e afetando todos os objetos já criados.

Naturalmente, não é proibido atribuir um valor a um membro estático na construtora. Mas neste caso **não** estamos falando de **inicialização** e sim de simples **atribuição**.

### *Exemplo de inicialização do membro estático:*

```
class Qualquer
{
    .....
public:
    static int MembroStat;
    .....
};
```

*agora o membro estático já pode ser inicializado:*

```
int Qualquer::MembroStat = 1;
// o operador de resolução de escopo indicando que MembroStat é membro
// da classe "Qualquer" -e não uma variável global.
```

*e o programa segue normalmente:*

```
int main( )
{
```

```

cout << Qualquer::MembroStat ; // RESULTADO = 1
    // não foi criado nenhum objeto
    // da classe "qualquer", mas o membro
    // estático pode ser usado a partir
    // do nome da classe e
    // do operador de resolução de escopo
    .....;
}

```

- A inicialização é obrigatória e é externa ao contexto de definição da classe.
- Logo, uma variável estática pode ser acessada mesmo antes da criação de qualquer objeto de sua classe, a partir do nome da classe e do operador de resolução de escopo.
- Por isso também deve ser evitado acessar um membro estático a partir de um objeto.
- Embora seja possível acessar um membro estático a partir de um objeto já existente, o melhor é **sempre** acessar o membro estático a partir da classe e não dos seus objetos.

```

int main( )
{
    Qualquer Um ;
    Qualquer Dois ;
    Um.MembroStat = 5 ;
    // POUCO CLARO: afinal, o objeto "dois" também será afetado.
    Qualquer::MembroStat = 5 ; // MELHOR: fica claro que foi
    // acessado um membro estático único na memória
}

```

### 7.5.7.2 • Funções-membro estáticas

- Um membro de dados estático pode ser acessado por qualquer função-membro, inclusive funções não estáticas.
- Mas uma função estática só pode acessar membros estáticos (funções estáticas não recebem o ponteiro **this** como parâmetro).
- E **devemos** acessar a função a partir do **nome da classe** e não de um objeto (pelos mesmos motivos levantados acima para acessar membros de dados estáticos).
- O melhor caminho para acessar membros de dados estáticos é utilizar funções estáticas, pois como essas funções não recebem o ponteiro **this** como parâmetro, através delas poderemos acessar os membros de dados estáticos mesmo que não tenha sido criado qualquer objeto da classe envolvida.

```

class Qualquer
{
    private:
        static int MembroStat ;

    public:
        int VarInt ;
        static int RetornaMembroStat( )
        {
            VarInt = 3 ; // ERRO: Função estática tentando
                        // acessar membro não estático
            return MEMBROSTAT; // OK: acessando membro estático.
        }
        static void GravaMembroStat( int Param )
        {
            MembroStat = Param ; // OK: acessando membro estático
        }
};

```



```

int Qualquer::MembroStat = 1 ; // inicialização do membro estático
int main( )
{
    Qualquer Um ;
    Um.GravaMembroStat ( 2 ) ;
    // CONFUSO: tentando acessar função-membro estática a partir
    // do objeto e não da classe.

    Qualquer::GravaMembroStat ( 2 ) // Melhor: a partir da classe.
}

```

## 7.5.8 ▪ Objetos como membros de classes.

### 7.5.8.1 ▪ Membros “objeto de outra classe”

- Já dissemos que não faria muito sentido estabelecer relações de herança que não existam na realidade.
  - Por exemplo, mesmo que a classe menu precise dos membros de data (para imprimir uma data na tela), não há relação de herança real entre uma data e um menu.
  - Além disso é preciso considerar também em **que medida** (ou em que quantidade) necessitamos dos membros de uma classe dentro de outra.
  - Vejamos o seguinte exemplo:

```

class Pagar
{
public:
    float Valor;
    Data DiaAtual ;
    Data DiaPagar;
};

class Receber
{
public:
    float Valor;
    Data DiaAtual ;
    Data DiaReceber;
};

int main( )
{
    Data dtHoje (1 , 1, 2011 );

    Pagar PG;
    PG.DiaAtual = dtHoje;
    std::cout << PG.DiaAtual.toString() << '\n';
    std::cout << PG.DiaPagar.toString() << '\n';

    Receber RC;
    RC.DiaAtual = dtHoje;
    std::cout << RC.DiaAtual.toString() << '\n';

    return 0;
}

```



No exemplo acima, tanto a classe pagar como receber possuem, cada uma, **dois membros da classe data**.

Se elas fossem herdeiras de data, além de estabelecer uma relação confusa (pois nem "pagar" nem "receber" *são datas*), não teriam o seu problema resolvido, pois *teriam apenas, cada uma, um único conjunto de membros da classe data, e no caso precisam de mais do que uma data*.



Nesse caso o melhor é que elas tenham objetos da classe data como membros (e quantos forem necessários) ao invés de utilizar herança

Nessa situação, caso um membro de dados "objeto de outra classe" tenha utilidade apenas em algumas situações menos frequentes, podemos também considerar a alternativa de utilizar ponteiros, alocando dinamicamente esses membros.

Assim só será alocada memória para os objetos caso (e quando) realmente for necessário.

### 7.5.8.2 • Membros "ponteiro para objeto de outra classe"

- Veja como ficaria a escrita das classes data, pagar e receber, usando ponteiros:

```
class Pagar
{
    public:
    float VALOR;
    Data * DiaAtual ;
    Data * DiaPagar;
};
class Receber
{
    public:
    float Valor ;
    Data * DiaAtual ;
    Data * DiaReceber ;
}
```



#### Atenção:

Veremos abaixo (função **main**) um exemplo de uso das classes "Pagar" e "Receber". Mas nesse exemplo existem **erros de acesso à memória**. Tente **localizar esses erros**:

```
int main( )
{
    // SUPONDO-SE que dia, mes e ano (de Data) fossem public:

    Data * pdtHoje = new Data (1,1, 2011); // Alocação de memória no heap
    pdtHoje->dia = 1 ; // USA O OPERADOR DE LIGAÇÃO A PONTEIRO
    pdtHoje->mes = 1;
    pdtHoje->ano = 1995;

    Pagar * pPG = new Pagar ; // PG É UM PONTEIRO
    Receber RC ;              // RC NÃO É UM PONTEIRO
```

```

pPG->DiaAtual= pdtHoje ; // ATENÇÃO aos operadores de
                          // LIGAÇÃO DE MEMBRO

RC.DiaAtual = pdtHoje ; // idem

pPG->DiaPagar->Dia = 10;
pPG->DiaPagar->Mes = 2;
PPG->DiaPagar->Ano = 1995;
std::cout << pPG->DiaPagar->toString() << '\n';

RC.DiaReceber->Dia = 10;
RC.DiaReceber->Mes = 2;
RC.DiaReceber->Ano = 1995;
std::cout << RC.DiaReceber->toString() << '\n' ;

}

```

### 7.5.8.3 • Membros “ponteiro para objeto da mesma classe”

Quando temos um membro ponteiro para objeto da mesma classe, e um objeto dessa classe é criado o C++ reservará apenas o **espaço para um ponteiro**.

Seria diferente se pretendessemos ter um objeto da classe como membro dela mesma: isto seria impossível pois o C++ precisaria alocar o objeto-membro (e para isso voltaria a consultar o molde que, novamente e mandaria criar o objeto-membro do objeto-membro ... E assim ao infinito.

Com um ponteiro, quebramos essa ação recursiva.


Será reservado espaço para um endereço que poderá mais tarde receber ou não o endereço real de um objeto criado através de **new**.

```

class Faixa
{
    float Maximo;
    float Minimo;

    Faixa * proxima ;
};

```

 **OBSERVE QUE:** quando as classes são diferentes **podemos** decidir, de acordo com a necessidade, se queremos um membro objeto da outra classe ou um membro ponteiro para esse objeto.



Contudo quando precisamos de um objeto da mesma classe como membro, é impossível usá-lo diretamente.

*Neste caso, só podemos ter um ponteiro para esse objeto.*

### 7.5.9 • Literais definidos pelo usuário (C++11)

O padrão **C++11** introduziu a capacidade de adicionar sufixos personalizados para literais, a fim de fornecer valores diferentes. Sufixos literais podem ser sobrecarregados de uma forma muito semelhante aos operadores.

Através dos literais definidos pelo usuário, alguns tipos de dados podem ter sua representação literal.

```
#include <iostream>

using namespace std;

long double operator"" _POW(long double BASE)
{
    return BASE * BASE;
}

int main()
{
    double x = 2.0_POW; cout << x << '\n';
    return 0;
}
```

Somente a lista de tipos abaixo é permitida para a criação de literais definidos pelo usuário.

const char *
unsigned long long
long double
char
wchar_t
char16_t
char32_t
const char *, std::size_t
const wchar_t *, std::size_t
const char16_t *, std::size_t
const char32_t *, std::size_t

O padrão **C++14** introduziu alguns literais definidos pelo usuário. Segue abaixo alguns dos principais:

- “s”, para criar std::basic\_string.
- “h”, “min”, “s”, “ms”, “us”, “ns”, para std::chrono::duration.

## 1. Exemplo

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

template <typename TIPO>
TIPO soma(TIPO a, TIPO b)
{
    return a + b;
}

int main()
{
    //Erro, pois as strings são para const char * e o tipo const char *
    //Não existe aqui uma sobrecarga do operador + para concatenação desse tipo.
    //cout << soma("Agit ", "Informatica") << '\n';

    //O literal definido pelo usuário "s", faz com que "Agit "
    //e "Informatica", sejam convertidos para std::basic_string.
    //O template de classe std::basic_string possui uma sobrecarga do operador +
    //para concatenar duas strings.
    cout << soma("Agit "s, "Informatica"s) << '\n';

    return 0;
}
```

## 7.5.10 ▪ Ambientes de nomes.

### 7.5.10.1 ▪ Evitando colisões de nomes.

Em C++ podemos isolar escopos através de ambientes de nomes, usando a palavra reservada **namespace**.

Um **namespace** irá impedir colisões de nomes de variáveis, funções ou classes escritas em módulos ou bibliotecas diferentes.

Isto significa que **namespace** é um recurso que visa reforçar o **encapsulamento** e, assim sendo, é mais um recurso disponível ao programador C++ para implementar a orientação a objetos.

Desse modo, se tivermos uma classe chamada “Data” e, algum dia, usarmos uma biblioteca que também tenha uma classe chamada “Data”, não haverá problemas **se** estivermos usando ambientes de nomes.

*Pois em C++ podemos fazer:*

```
namespace agit
{
    class Data;
};

e agora a nossa classe Data será declarada assim:

class agit::Data
{
    private:
        .....
    public:
        .....
};
```

e usaremos a classe assim

```
int main( )
{
    agit::Data Hoje;
    .....
    return 0;
}
```

E, para fins de simplificação, em um determinado escopo, podemos declarar o **namespace** em uso, o que irá permitir que usemos nomes declarados dentro dele sem referência explícita ao **namespace**. Assim:

```
using namespace agit;
Data Hoje; // OK;
```



Mas cuidado! Ao **anular** um namespace você perde o benefício da **proteção** dos nomes de símbolos declarados dentro do namespace. Isso torna possível **colisões** com nomes idênticos de símbolos diferentes.

### 7.5.10.2 • Usando namespace para organizar conjuntos de software.

Além disso, **namespace's** também permitem organizar um projeto ou uma biblioteca em um conjunto de unidades lógicas, onde cada uma delas é representada por um **namespace**.



Ou seja: o namespace permite que evitemos **colisões** com bibliotecas de terceiros e permite também que seja claramente exposta a **organização lógica** de uma biblioteca ou conjunto de software.

**Por exemplo:**

```
namespace agit
{
    namespace tipos // tipos básicos
    {
        class Data;
        class Moeda;
    };
    namespace financ // financeiro
    {
        class Pagar;
        class Receber;
        class Impressao;
    };
    namespace admin // administrativo
    {
        class Controle;
        class Processos;
        class Impressao;
    };
}; // Fim da declaração do namespace agit.
```

```
// - declarar as classes aqui...
int main()
{
    agit::tipos::Data hoje;
    agit::financ::Pagar pag;
    agit::admin::Controle ctl;
    agit::financ::Impressao fImpr;
    agit::admin::Impressao aImpr;
    return 0;
}
```

Assim, através de **namespace**'s aninhados, temos claramente exposta a organização lógica do conjunto "*agit*", com a especificação dos subconjuntos e a posição das classes dentro deles.

Além disso evitamos colisão entre símbolos de diferentes conjuntos da PRÓPRIA biblioteca, como é o caso das duas classes de impressão:

```
agit::financ::Impressao
agit::admin::Impressao
```

E se quisermos simplificar a escrita, podemos usar(em qualquer escopo) o "**using namespace**" ou então criar **sinônimos** simplificadores:

```
namespace agFin = agit::Financ;
```

### 7.5.10.3 • O namespace "std".

Toda a biblioteca padrão C++ está declarada dentro do namespace "std". Isto significa que, na verdade, o nosso tipo "**string**" é:

```
std::string
```

e o nosso "**cout**" é:

```
std::cout.
```

Contudo, se declararmos:

```
using namespace std;
```

poderemos agora usar "**string**" e "**cout**" sem fazer referência ao seu namespace.

O que causa alguma confusão é que encontramos muito código onde a declaração "**using namespace std**" não está presente e mesmo assim os recursos da biblioteca padrão são usados sem qualquer referência ao "**std**".

Isto se deve ao seguinte: os arquivos de inclusão da biblioteca padrão **não** têm a extensão ".h".

Desse modo, usando a biblioteca padrão, fazemos:

```
#include <iostream> // sem .h
#include <string> // sem .h
```

E, aí, temos duas alternativas:

*OU*

```
std::string MinhaString;
std::cout << std::endl;
```

*OU*

```
using namespace std;
string MinhaString;
cout << endl;
```

Contudo, os compiladores fornecem também um segundo jogo de arquivos de inclusão COM a extensão **".h"**.



Esses arquivos são obsoletos. São mantidos por compatibilidade com C++ anterior ao padrão C++98.

Por isso, **se**, ao invés de

```
#include <iostream> // SEM o .h
```

usarmos

```
#include <iostream.h> // COM o .h
```

então usaremos diretamente

```
cout << endl; // não tínhamos "namespaces" no C++ antigo.
```



Use sempre os arquivos atuais, coerentes com o padrão.

#### 7.5.10.4 • Namespace inline (C++11)

No padrão C++11 foram introduzidos os namespaces inline que podem auxiliar na evolução das bibliotecas através de um mecanismo de versionamento.

```
#include <iostream>
```

```
using namespace std;
```

```
//arquivo Math_V_2_0.h:
```

```
inline namespace Math_V_2_0
```

```
{
```

```
    //Corrigido o problema da divisao por 0
```

```
    double divisao(double a, double b)
```

```
    {
```

```
        return b != 0 ? a / b : 0;
```

```
    }
```

```
    //Novo recurso
```

```
    double soma(double a, double b)
```

```
    {
```

```
        return a + b;
```

```
    }
```

```
}
```

```
//arquivo Math_V_1_0.h:
```

```
namespace Math_V_1_0
```

```
{
```

```
    double divisao(double a, double b)
```

```
    {
```

```
        return a / b;
```

```
    }
```

```
}
```

```
int main()
```

```
{
```

```
    cout << Math_V_1_0::divisao(10, 2) << '\n'; //Versão antiga
```

```
    cout << Math_V_2_0::divisao(10, 2) << '\n'; //Versão nova
```

```
    cout << divisao(10, 2) << '\n'; //Versão nova por default
```

```
    return 0;
```



```
}
```

### 7.5.10.5 • Exemplos de uso de namespace.

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

```
// OBS: não estamos usando a linha abaixo:
// using namespace std;
```

```
/* =====
```

Neste exemplo, iremos explorar alguns conselhos do criador de C++, Bjarne Stroustrup:

- 1) "Use ambientes de nomes (namespace's) para expressar uma estrutura lógica".  
Exemplo:

```
namespace Financeiro
{
    class Pagar;
    class Receber;
};
```

- 2) "Coloque cada nome não-local em um namespace". Exemplo:

*Variáveis globais do projeto Contas a Pagar:*

```
namespace ConPag_Glob
{
    int ParametroGeral=0;
};
```

- 3) "Projete um ambiente de nomes de modo a não acessar acidentalmente um ambiente não relacionado"

*Exemplo: O nome "financeiro" poderia existir em um outro ambiente, pois é muito comum. Uma solução seria:*

```
namespace agit
{
    namespace financeiro
    {
        class Pagar;
        class Receber;
    };
    namespace admin
    {
        class Patrimonio;
        class ControlaProcessos;
    };
};
```

*Agora, "financeiro" tornou-se: "agit::financeiro"*

*E "admin" deve ser acessado como: "agit::admin"*

- 4) "Evite nomes muito curtos para ambientes de nomes"

Nomes muito curtos facilitam colisões com outros nomes.

Contudo, se usarmos, como acima, nomes aninhados, esse problema será minimizado, pois cada nome só faz sentido dentro do ambiente em que foi declarado.

Assim o nome mais externo ("agit", no exemplo) garantirá que os seus nomes internos estarão isolados com relação a outros ambientes de nomes.

Agora, bastará garantir que não teremos um outro ambiente de nomes onde o nome principal seja, também, "agit".

5) "Para simplificar, se for necessário, use sinônimos para ambientes de nomes"

*Exemplo:*

```
namespace agfin = agit::financeiro;
```

6) "Use 'using namespace' somente para migração ou em um escopo local."

*E este conselho também poderia ser aplicado a:*

*"so crie sinônimos em um escopo local".*

O "using namespace" ou os sinônimos, devem ser usados preferencialmente apenas dentro de uma determinada função ou classe.



Do contrário, o namespace, na prática, estará sendo anulado.

```
*/
```

```
// ===== Declara o namespace global "agit" que envolverá vários conjuntos lógicos:
```

```
namespace agit
```

```
{
```

```
    class Data;
```

```
    class Outra;
```

```
    class etc;
```

```
    namespace Financ
```

```
    {
```

```
        class Pagar;
```

```
        class Receber;
```

```
        class Impressao;
```

```
    };
```

```
    namespace admin
```

```
    {
```

```
        class Controle;
```

```
        class Processos;
```

```
        class Impressao;
```

```
    };
```

```
};
```

```
// ===== Declaração das classes:
```

```
class agit::Data
```

```
{
```

```
    public:
```

```
        char m_Dia;
```

```
        char m_Mes;
```

```
        short m_Ano;
```

```
};
```

```
class agit::Outra
```

```
{
```

```
    public:
```

```
        int m_iOutro;
```

```
};
```

```
class agit::etc
```

```
{
```

```
    public:
```

```
        int m_iEtc;
```

```
};
```

```
class agit::Financ::Pagar
```

```
{
```

```

    public:
        double m_Pagar;
};
class agit::Financ::Receber
{
    public:
        double m_Receber;
};
class agit::Financ::Impressao
{
    public:
        Impressao(){}
        double m_Receber;
};
class agit::admin::Impressao
{
    public:
        Impressao(){}
        double m_Receber;
};

// namespace para variáveis globais de um determinado projeto.
// Por exemplo o projeto "Contas a Pagar":
namespace ConPag_Glob
{
    int ParametroGeral=0;
};
// contudo, no exemplo acima, talvez fosse melhor criar uma classe
// "ConPag_Glob",
// onde a variável seria private e seria acessada apenas através de funções.

int main()
{
    std::cout << "Testando NameSpace" << std::endl;

    agit::Financ::Pagar Pag;
    Pag.m_Pagar = 101.50;

    agit::Data hoje;
    hoje.m_Dia = 3;

    // ERRO:
    // Data amanha; // não indicou o namespace
    using namespace agit;
    Data amanha; // Agora sim !!!:
    amanha.m_Dia = 10;

    // ERRO:
    // Receber Rec; // não indicou o namespace
    using namespace agit::Financ;
    Receber Rec; // Agora sim !!! :
    Rec.m_Receber = 10.33;

    ConPag_Glob::ParametroGeral = 10;

    // ERRO:
    // std::cout << ParametroGeral << std::endl;
    // namespace de ParametroGeral foi omitido

    // Agora sim:
    std::cout << ConPag_Glob::ParametroGeral << std::endl;
    // ou então:

```

```

using namespace ConPag_Glob;
std::cout << ParametroGeral << std::endl;
    // OK: namespace em uso.

// criando um sinônimo local:
namespace agFin = agit::Financ;
agFin::Pagar Pag2;
Pag2.m_Pagar = 44.3;
// apenas a partir daqui, até o fim da função,
    é que o std poderá ser suprimido:
using namespace std;
cout << Pag2.m_Pagar << endl;

// usando namespace evitamos o conflito
    entre as duas classes de impressão:
agit::admin::Impressao aImpr;
agit::Financ::Impressao fImpr;

return 0;
}

void Funcao()
{

```



Todo o código abaixo está errado:

```

/*
    agFin::Pagar Pag2;
    // o sinônimo agFin só existe em main !

    Pag2.m_Pagar = 44.3;

    cout << Pag2.m_Pagar << endl;
        // teria que usar std::cout e std::endl
        // pois o "using namespace std;", feito acima,
        // foi feito dentro de main
        // (e, portanto, é um uso local pertencente a main):
*/
}

```

## 7.6 • Questões para revisão do capítulo 7

Responda às questões abaixo. **Em seguida, compare suas respostas** com as respostas localizadas no **Anexo B-7, página 471**. Caso não entenda, encaminhe as dúvidas ao instrutor.

- porque as **estruturas (struct ou class)** do **C++** são **melhores** que as estruturas **(struct)** do C?

- o que são as restrições de acesso **private** e **public** ?

---

---

---

- o que significa "**this**" ?

---

---

---

- o que são funções **operadoras** ?

---

---

---


- o que significa "**const**" na definição do tipo de um parâmetro de função?

---

---

---

- responda à pergunta que está no comentário do código abaixo:


```
class X
{
    X( )  // Que função é esta e para que serve?
    {
        .....
    }
    .....
};
```

---

---

---

- responda à pergunta que está no comentário do código abaixo:

```
class X
{
    ~X( )  // Que função é esta e para que serve?
    {
    }
};
```

---

---

---

- 
- responda às 4 perguntas que estão nos comentários do código abaixo:

```
class X
{
    bool operator < ( const X & obj_2 ) const ;

    //      1- Que função é esta e para que serve?
    //      2- Por que a palavra "const" aparece aí duas vezes?
    //      3- O que faz aí o "e-comercial" (&) ?
    //      4- O que faz aí a letra "X" ?
};
```

---

---

---

---

---



## • Capítulo 8

### ▪ Programação genérica

---

8.1 •	Templates.....	265
8.2 •	Templates de função.....	265
8.3 •	Templates de classes.....	268
8.4 •	Templates com quantidade variável de argumentos (C++11).....	270



## 8.1 • Templates



Em algumas situações temos uma mesma lógica que se aplica a diversos tipos. Nesses casos, **o foco não é o tipo e sim o algoritmo**.

## 8.2 • Templates de função

Por exemplo: se quisermos escrever uma função que calcule o menor entre dois valores, teremos um algoritmo que servirá para **qualquer tipo** que disponha do **operador relacional *menor que***. Contudo, uma **função** exige que seus parâmetros e seu retorno tenham tipos **explicitamente** declarados:

```
int Minimo ( int a , int b)
{
    return ( ( a < b ) ? a : b );
}
```

Agora, se quiséssemos calcular o menor valor entre dois valores do tipo **double**, teríamos que escrever uma nova função:

```
double Minimo ( double a , double b)
{
    return ( ( a < b ) ? a : b );
}
```

O que fizemos foi “copiar” a primeira função, para depois “colar” em outro lugar e finalmente “trocar” **int** por **double**. Isso é trabalho de máquina, não de gente.

Não seria possível que o compilador fizesse isso por nós? Não podemos pedir a ele que “copie”, “cole” e “troque”?

Sim, podemos. E isto é feito através de “gabaritos”(templates). Em um **template**, ao invés de indicar um tipo real, indicamos que ali, **futuramente**(no momento em que formos **usar** a função) **será informado um tipo** de acordo com as necessidades:



**Ao declará-lo, precisamos indicar que vamos escrever um template de função e não uma função:**

```
template <class TIPO_PARAMETRIZADO>
// tipo de retorno
TIPO_PARAMETRIZADO Minimo(
    TIPO_PARAMETRIZADO a // tipo do parâmetro 1
    TIPO_PARAMETRIZADO b ) // tipo do parâmetro 2
{
    return ( a < b ) ? a : b;
}
```



Se, em algum momento, usarmos esse **template** para trabalhar com dois valores do tipo **int**, o *compilador* criará uma **função** onde **TIPO\_PARAMETRIZADO** será **substituído por int**.

E, se, em algum outro momento, usarmos esse template para trabalhar com dois valores do tipo **double**, o *compilador* criará uma **outra função** onde **TIPO\_PARAMETRIZADO** será **substituído por double**.

Logicamente essa operação poderá ser feita **inline**, evitando uma chamada de função desnecessária, já que o código é muito pequeno. Um *template de função tanto pode ser instanciado “inline” ou como uma função comum*.

*Templates* são assim modelos para que o compilador escreva funções e classes baseadas em tipos que serão passados como parâmetros de acordo com a necessidade de cada momento. Por isso os *templates* também são chamados de *recursos*(funções ou classes) *com tipos parametrizados*.

Assim, na implementação de um algoritmo genérico, ao invés de criar uma classe para cada tipo de dados, ou então uma classe com diversas funções sobrecarregadas, uma para cada tipo desejado, podemos criar um único esqueleto de classe capaz de operar com diversos tipos de dados - pois os tipos serão passados como parâmetros. E o mesmo pode ser feito com funções globais.



Um programador **C** poderia argumentar que isso tudo poderia ser feito através da diretiva de pre-processamento **#define**.

É verdade. Mas teríamos aí vários inconvenientes.

Vejamos porque.

### EXEMPLO:

Utilizando o exemplo acima do cálculo do menor valor (minimo), teríamos as seguintes alternativas:

#### 1) NÃO usando TEMPLATES:

##### 1.A) usando MACROS:

```
#define MINIMO(x,y) ((x)<(y)?x:y)
```

```
// que seria usado assim:
```

```
a = MINIMO(2,3);
```

```
// que seria trocado para
```

```
// a = ((2)<(3)?2:3);
```

A vantagem deste método seria a simplicidade e uma baixa sobrecarga de código já que ocorreria apenas a substituição da pseudo-função(ou macro) por uma simples operação.

A desvantagem é que não ocorrerá checagem dos tipos de x e y.

Se os tipos forem comparáveis, (como um int ou um double) o compilador não emitirá mensagens de erro e nem mesmo uma "warning".

Se não forem (comparação entre duas estruturas, não providas de operador de comparação compatível) o erro apresentado pelo compilador indica a linha onde a macro é usada mas não fará qualquer referência à própria macro (já que o código foi expandido no pre-processamento e o compilador não enxerga mais a macro).

A mensagem será incompleta, e muitas vezes confusa.

Já em um template, a mensagem de erro fará referência explícita ao próprio template, ainda que atualmente os compiladores tenham dificuldade de nomear os erros em situações mais complexas.

Além disso, usando macros teremos problemas adicionais, devido ao fato de que o pre-processador realiza uma simples troca sem qualquer consideração de qualidade.

Por isso, repare que o seguinte uso da macro MINIMO, acarretaria perda de performance:

```
a = MINIMO ( sqrt( x ) * 2, sqrt( y ) / 2 );
// seria traduzido para
a = ( (( sqrt( x ) * 2 ) < ( sqrt( y ) / 2 ) )
      ? sqrt( x ) * 2 : sqrt( y ) / 2 );
```

E assim operações lentas (raiz quadrada e multiplicação ou divisão) seriam executadas duas vezes (uma vez na comparação e outra na decisão).

E essa forma de expansão do código pode acarretar problemas bem piores quando utilizamos operadores compostos ou o incremento e o decremento (que realizam uma operação aritmética seguida de uma operação de atribuição).

Ou seja, nunca deveríamos usar esses operadores ( ++, --, +=, -=, etc. ) em macros. Vejamos o que pode ocorrer:

```
int x=6, y=6;
cout << MINIMO( ++x , y ) << endl;

a linha acima seria expandida para
cout << ( ( ++x ) > ( y ) ) ? ( ++x ) : ( y ) << endl ;
```

**Ou seja:** quem usou a macro como se fosse uma função provavelmente espera que a operação de incremento em “x” seja realizada uma **única** vez.

Contudo, com a expansão, o incremento poderá ser efetuado **duas** vezes (sempre que ++x resultar maior que y).

E, se o objetivo fosse realmente esse, semelhante código seria altamente **confuso**.

### 1.B) usando FUNÇÕES SOBRECARGADAS:

```
inline int Minimo( int a, int b )
{ return ( a < b ) ? a : b; }

inline long Minimo( long a, long b )
{ return ( a < b ) ? a : b; }
```

*(e mais funções para char, double, unsigned int, unsigned long, etc...)*

A operação utilizará os tipos corretamente pois os **parâmetros são tipificados** e também o **retorno** da função, indicando assim tanto o que pode ser enviado como o que pode ser esperado (e eventualmente armazenado) como resultado da operação.

O compilador fará a checagem normalmente e emitirá as mensagens apropriadas.

A desvantagem é que teremos muitas funções, gerando uma sobrecarga de código a ser mantido.

## 2) USANDO TEMPLATES:

Usando templates, não haveria necessidade de toda essa repetição de código:

```
// para identificar um parâmetro template
// podemos usar "class" ou "typename"
// a função template pode ser inline ou não, dependendo do caso,

template <typename TIPO_PARAM>
TIPO_PARAM inline Minimo( TIPO_PARAM a, TIPO_PARAM b )
{
    return ( a < b ) ? a : b;
}
```

*que poderia ser usado assim:*

```
int X=5 , Y=5;
double A=3.4 , B=5.6;

a = Minimo <double> ( A, X ); //TIPO_PARAM será double
a = Minimo <int>(X,Y);          // TIPO_PARAM será int
a = Minimo ( X , Y);           // TIPO_PARAM será int
                                // já que os tipos de x e y estão claros

a = Minimo( A, (double)X);     // TIPO_PARAM será double
                                // já que A é double e X foi convertido para double


cout << Minimo( X , Y ) << endl; // o TIPO Parametrizado será int
cout << Minimo( A , B ) << endl; // o TIPO Parametrizado será double
```

E o template será usado para substituir cada linha de operação do modo adequado.

A vantagem sobre as funções sobrecarregadas é que reduzimos código sem reduzir a flexibilidade que temos com a sobrecarga.

E a vantagem sobre a macro é que isso foi feito sem reduzir a checagem segura de tipos oferecida pelo compilador e sem correr o risco de duplicar operações compostas, ou outras expansões indesejadas.

---

 Mas como a linguagem permite a **sobrecarga de operadores**, várias classes podem ter o "**operator <**", usado por Minimo. **É o caso da nossa class Data.** Portanto "Minimo" poderá ser usado por qualquer tipo que tenha o "**operator <**". Passar **estruturas por cópia pode ter um custo alto**, dependendo do caso.

---

Então, o correto seria usar referências (const ou não-const, dependendo do caso):

```
template <typename TIPO>
const TIPO & inline Minimo( const TIPO & a, const TIPO & b )
{
    return ( a < b ) ? a : b;
}
```

Então podemos fazer:

```
Data dt1(1,1,2000) , dt2(2,1, 2000);
const Data & dt3 = Minimo( dt1, dt2);
std::cout << dt3.toString() << '\n' ; // vai imprimir 01/01/2000
```

## 8.3 • Templates de classes

```
#include <iostream>
using namespace std; // cuidado com isso...
```

```
// TEMPLATE de CLASSE "ListaDeValores":
template <class TIPO, size_t dim> class ListaDeValores
{
    private:
        TIPO buffer[dim];
        // membro de dados do TIPO PARAMETRIZADO
        // com a dimensão constante "dim"

    public:
        void atribui( size_t Indice , TIPO valor );
        // segundo parâmetro: do TIPO PARAMETRIZADO

        TIPO retorna(size_t Indice );
        // valor de retorno: do TIPO PARAMETRIZADO
};

// usar assim, por exemplo:
// ListaDeValores <int, 10> MinhaLista;

// Template de função-membro "atribui" do template de classe "ListaDeValores":
template <typename TIPO, size_t dim>
void ListaDeValores<TIPO, dim>::atribui(size_t Indice, TIPO valor)
{
    buffer[Indice] = valor;
}

// Template de função-membro "retorna" do template de classe
// "ListaDeValores":
template <typename TIPO, size_t dim>
TIPO ListaDeValores<TIPO, dim>::retorna(size_t indice)
{
    return buffer[indice];
}
```

- **OBS:** o template "atribui" tem retorno void e recebe como argumentos-template:
  - um valor do tipo size\_t
  - um valor do tipo parametrizado "TIPO"
- Já o template "retorna" tem como retorno
  - um valor do tipo parametrizado "TIPO"
  - e recebe como argumento-template um valor do tipo size\_t.
- Desse modo, o tipo parametrizado "TIPO" tanto pode servir para definir os tipos de membros de dados do Template de Classe como também os tipos de retornos e parâmetros das funções-membro..

```
template <class TRet, class TArg>
TRet SomaValores( const TArg & a, const TArg & b )
{
    return ( a + b );
}
```

O template acima precisa que seu primeiro parâmetro template (TRet) seja explicitado no uso, já que o retorno não pode ser deduzido.

```
int a = 5, b=6;
int c = SomaValores<int, int> (a,b );

// explicitou TRet e TArg. Mas TArg pode ser deduzido
// (pois os tipos de a e b estão claros); Então basta explicitar TRet:
c = SomaValores<int>(a,b);
```

```

int main()
{
    // usa ListaDeValores para uma MATRIZ DE 10 valores do tipo char
    ListaDeValores<char, 10> ob1;
    cout << "=== Testa template 'ListaDeValores' com CHAR:" << '\n';
    for ( int iC = 0; iC < 10; iC++ )
    {
        ob1.Atribui( iC, 'A'+iC );
        cout << ob1.Retorna(iC) << ( (iC<9) ? " , " : "" ) ;
    }
    /** Resultado: A , B , C , D , E , F , G , H , I , J

    // usa ListaDeValores para uma MATRIZ DE 6 valores do tipo int
    ListaDeValores<int, 6> ob2;
    cout << "\n\n=== Testa template 'ListaDeValores' com INT:"
                                                << '\n';

    for ( size_t c = 0; c < 6; ++c+)
    {
        ob2.atribui( c, 1+static_cast<int>(c) );
        cout << ob2.retorna(c) << " , " ;
    }
    /** Resultado: 1 , 2 , 3 , 4 , 5 , 6,

    x=2; y=5;
    cout << Minimo( ++x , y ) << '\n'; // valores x e y do tipo INT
    /** Resultado: 3 (int)
    // MAS se Minimo fosse uma Macro o resultado seria 4 (incorreto)

    cout << "\n===Usando o mesmo template para tipos DOUBLE:"
                                                << '\n';

    double a=5.1 , b=5.2 ;
    cout << Minimo( a , b ) << '\n'; // valores a e b do tipo DOUBLE
    /** Resultado: 5.1 (double)

    cout << "\n===Pode ser usado tambem com "
                                                << "definicao EXPLICITA do tipo:" << '\n';
    cout << Minimo <int>( x , y ) << '\n';
    /** Resultado: 3 (int)
    cout << Minimo<double>( x , y ) << '\n';
                                                // x e y convertidos para double
    /** Resultado: 3 (double)

    cout << "\n===Exemplo de um template que recebe dois tipos\n";
    cout << SomaValores <int>( x,y ) << '\n';

    return 0;
}

```

## 8.4 • Templates com quantidade variável de argumentos (C++11)

Os templates agora podem ter uma quantidade variável de argumentos:

```
template<typename ...Args> class Sample;
```

```
template<typename T, typename ...Args> T funcao(Args ...args);
```

Em uma lista de parâmetros de função, o parâmetro a ser expandido deve aparecer por último.

Geralmente, a manipulação de um template variadic é feita através de recursividade, onde se analisa o primeiro parâmetro e passa o resto dos argumentos para o template novamente.

As únicas operações que podem ser feitas sobre a lista de parâmetros variáveis são (...) para se referir à lista e sizeof...() para descobrir a quantidade de elementos na lista.

## 1. Exemplo:

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

//Conta número de parâmetros do template
template <typename ...A>
int func(A... arg)
{
    return sizeof...(arg);
}

int main(void)
{
    cout << func(1,2,3,4,5,6) << '\n';
    return 0;
}
```

## 2. Exemplo:

Essa declaração pode parecer estranha. Isso porque os argumentos de template variadics são percorridos de forma recursiva, em vez de forma iterativa. Abordamos o primeiro argumento da função e depois uma chamada a função usando o resto dos argumentos. Observe como nós definimos duas sobrecargas de função aqui. Uma das sobrecargas de função não tem argumentos porque C++ precisa saber como terminar a recursão. Se essa sobrecarga não for especificada, então vai ocorrer um erro de compilação.

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

//Repare que para capturarmos o parâmetro real da função fomos
//obrigados a criar uma sobrecarga da função newPrintf e se retirarmos
// void newPrintf(){cout << '\n';} o programa não compila.
void newPrintf(){cout << '\n';}

template <typename Valor, typename ...Parametros>
```

```
void newPrintf(const Valor &cabeca, const Parametros &...calda)
{
    cout << cabeca;

    //Chama a própria função passando parâmetro por parâmetro capturado.
    //Quando chegar ao último parâmetro, o C++ precisa encontrar uma função
    //com mesmo nome que não leve parâmetros para saber onde tudo
    //termina - caso contrário teremos erro de compilação.
    newPrintf(calda...);
}

int main()
{
    newPrintf("A soma de 10 com 20 eh: ", (10 + 20),
        "\nA divisão de 19 por 2 eh: ", (19.0 / 2.0));

    return 0;
}
```



---

## • Capítulo 9

### ▪ A biblioteca padrão

---

9.1 •	std::string.....	276
9.2 •	std::vector.....	277
9.3 •	std::list.....	277
9.4 •	std::map.....	279
9.5 •	std::vector < std::vector < > >.....	280
9.6 •	Laço <i>for</i> para <i>containers</i> (C++11).....	281
9.7 •	Outras classes containers.....	282
9.7.1 ▪	initializer_list (C++11).....	282
9.7.2 ▪	forward_list (C++11).....	283
9.7.3 ▪	unordered_map (C++11).....	285
9.7.4 ▪	unordered_multimap (C++11).....	285
9.7.5 ▪	unordered_set (C++11).....	286
9.7.6 ▪	unordered_multiset (C++11).....	286
9.7.7 ▪	array (C++11).....	287
9.7.8 ▪	tuple (C++11).....	287

Além das funções herdadas do C e das classes de *stream* para operações de entrada/saída, a biblioteca padrão oferece a **Standard Template Library (STL)** que é uma biblioteca de **templates** que nos permitem criar, diretamente, **novos tipos de dados**.

Os **templates** de maior aplicação prática são:

- `basic_string`,
- `vector`,
- `list`,
- `map` e `multimap`
- `set` e `multiset`

Além disso dois tipos já são fornecidos:

- `string`; que é um sinônimo para `basic_string<char>`  
( `typedef basic_string<char> string` );
- `wstring`; que é um sinônimo para `basic_string<wchar_t>`  
( `typedef basic_string<wchar_t> wstring` ), sendo este último usado para UNICODE.

### Descrição:

`string`:

Esta classe permite encapsular uma matriz de caracteres (tipo `char`) com tamanho variável e redimensionamento automático.

Isto porque as operações de alocação e liberação de memória são garantidas pela classe, que realiza tais atividades de modo otimizado e seguro.

`wstring`:

Esta classe permite encapsular uma matriz de caracteres sob o padrão *UNICODE* (tipo `wchar_t`, o qual é implementado como um sinônimo para *unsigned short*).

As observações feitas acima (para *string*) aplicam-se igualmente aqui.

`vector`:

Este gabarito de classe permite encapsular uma sequência com quantidade variável de elementos de um tipo especificado.

A sequência de elementos é armazenada como uma *array*.

`valarray` :

De modo semelhante a *vector* este gabarito de classe permite encapsular uma sequência com quantidade variável de elementos de um tipo especificado.

A sequência de elementos é armazenada como uma *array*.

O gabarito *valarray* é diferente de *vector* por duas razões:

1) *valarray* implementa diversas operações aritméticas entre elementos correspondentes de objetos *valarray* do mesmo tipo e tamanho;

*Exemplo:*

`x = cos(y) + sin(z),`

será válido se `x`, `y` e `z` forem do tipo `valarray<int, _mesmo_tamanho>`

2) *valarray* define diversos modos para acesso aos seus elementos através de índice, sobrecarregando o operador `[]`.

`list`:

Este gabarito de classe permite encapsular uma sequência com quantidade variável de elementos de um tipo especificado

A sequência de elementos é armazenada como uma lista ligada bidirecional.

`map`:

Este gabarito de classe permite encapsular uma sequência com quantidade variável de elementos do tipo *pair* `<const Key, T>`, sendo que o primeiro elemento de cada par é a chave de ordenação e o segundo elemento é o seu valor associado.

A sequência é organizada de modo a permitir que busca, inserção e remoção de um elemento sejam efetuadas rapidamente (o número de operações envolvidas é proporcional ao logaritmo do número de elementos na sequência)

- ✂ Além desses e outros **containers** a STL contém **algoritmos** genéricos e objetos função (**functors**).
- ✂ Veja em [www.cplusplus.com](http://www.cplusplus.com) ou em [www.cppreference.com](http://www.cppreference.com)
- ✂ Além disso, nos exercícios do curso vários deles serão exemplificados.

### *Exemplos de uso:*

```
#ifdef WIN32
#pragma warning(disable:4786) // específico Microsoft.
#endif

#include <iostream>
#include <string>
#include <vector>
#include <list>
#include <map>

using namespace std;

// cria um sinônimo para um vector de int's:
typedef vector<int> vector_int;
// cria um sinônimo para um list de int's:
typedef list<int> list_int;
// cria um sinônimo para um map onde a chave é int e o valor associado é string:
typedef map<int, string> map_int_string;
// cria um sinônimo para um vector onde cada elemento é um vector de int's:
typedef vector<vector_int> vector2_int;

void TestaString();
void TestaVector();
void TestaList();
void TestaMap();
void TestaVectorVector();

void Pausa();

class FaixaImposto // mais abaixo faremos um list desta struct
{
public:
    float m_Min;
    float m_Max;
    float m_Percent;

    // construtor:
    FaixaImposto ( float Min, float Max, float Percent )
    {
        m_Min = Min ;
        m_Max = Max ;
        m_Percent = Percent;
    }
};

// cria um sinônimo para uma lista de FaixaImposto (list<FaixaImposto>)
typedef list<FaixaImposto> list_faiximp;

int main()
```

```

{
    TestaString();
    TestaVector();
    TestaList();
    TestaMap();
    TestaVectorVector();
    return 0;
}

```

## 9.1 • std::string

```

// testa string
void TestaString()
{
    cout << "testando string" << endl << endl;
    string texto;
    texto = "ABC"; // operador de atribuição: realiza uma cópia
    cout << texto << endl;
    cout << texto.length() << endl;
    // quantidade de caracteres até o terminador zero
    cout << texto.max_size() << endl;
    // máximo de caracteres que pode ser alocado

    texto += "DEF"; // operador "+=": concatenação
    cout << texto << endl;

    string subtexto = texto.substr(2,3);
    // extraindo uma substring (um segmento)
    cout << subtexto << endl;

    int posic = texto.find('E'); // busca o caracter 'E' na string texto
    if ( posic != string::npos )
        // "string::npos" = "null position" (não localizado)
        cout << posic << endl;

    posic = texto.find("DE"); // busca a string "DE" na string texto
    if ( posic != string::npos )
        // "string::npos" = "null position" (não localizado)
        cout << posic << endl;

    texto.erase(); // " apaga" todos os caracteres
    // (na verdade, termina a string no primeiro byte)
    if ( texto.empty() ) // agora, verifica se realmente está vazia
        cout << "string vazia" << endl;

    texto = "ARI";
    cout << texto << endl;
    texto[0]='E'; // altera o primeiro caracter
    cout << texto << endl;
    texto[1]='L'; // altera o segundo caracter
    cout << texto << endl;
    texto[2]='A'; // altera o terceiro caracter
    cout << texto << endl;

    Pausa();
}

```

## 9.2 • std::vector

```
// ===== Usando vector:
void TestaVector()
{
    cout << "VECTOR: Inicia teste:" << endl << endl;

    //o vetor abaixo será criado com zero elementos:
    vector<int> VecInt;

    // Adicionar um elemento ao FINAL do vetor
    // contendo o valor inteiro "10":
    VecInt.push_back(10) ;

    // Vejamos agora como está o vetor:
    cout << "Tamanho de VecInt : " << VecInt.size() << endl;
    cout << "Tamanho maximo de VecInt : " << VecInt.max_size()
                                                << endl;

    // preenche CINCO elementos da lista com o número 11
    VecInt.assign( 5 , 11);

    // Vejamos agora como ficou o vetor:
    cout << "\nSituacao de VecInt apos preenchimento de 5 elementos"
                                                << endl;

    cout << "Tamanho de VecInt : " << VecInt.size() << endl;
    cout << "Tamanho maximo de VecInt : " << VecInt.max_size()
                                                << endl;

    cout << "Impressao dos elementos ate size()" << endl;
    int iC;
    for ( iC= 0; iC < VecInt.size(); iC++ )
        cout << VecInt[iC] << " - ";
    cout << endl;

    VecInt[2] = 12;
    cout << "Alterou o valor do TERCEIRO elemento para 12" << endl;
    for ( iC= 0; iC < VecInt.size(); iC++ )
        cout << VecInt[iC] << " - ";
    cout << endl;

    // REDIMENSIONA espaco para 200 elementos:
    VecInt.resize(200);

    // Vejamos agora como ficou o vetor:
    cout << endl << "Situacao de VecInt apos REDIMENSIONAMENTO "
        "para 200 elementos:" << endl;

    cout << "Tamanho de VecInt : " << VecInt.size() << endl;
    cout << "Tamanho maximo de VecInt : " << VecInt.max_size()
                                                << endl;

    Pausa();
}
```

## 9.3 • std::list

```
// ===== Usando list:
void TestaList()
{
    cout << "LIST: Inicia teste:" << endl << endl;

    list<int> ListInt;
```

```

list_int::iterator ListIntIt;
// Insere no inicio;
ListInt.insert (ListInt.begin(), 1);
ListInt.insert (ListInt.begin(), 2);
// Insere no fim:
ListInt.insert (ListInt.end(), 3);
// Imprime a lista:
for ( ListIntIt = ListInt.begin(); ListIntIt != ListInt.end(); ++ListIntIt)
{
    cout << *ListIntIt << " - ";
}
cout << endl;
// Insere 3 vezes o número 4 no fim da lista:
ListInt.insert (ListInt.end(), 3, 4);
// Reimprime
for (ListIntIt = ListInt.begin(); ListIntIt != ListInt.end(); ++ListIntIt)
{
    cout << *ListIntIt << " - ";
}
cout << endl;
// uma lista do tipo list_faiximp (class list_faiximp)
list_faiximp ListFaixImp;
list_faiximp::iterator ListFaixImpIt;
// Insere no fim:
ListFaixImp.insert ( ListFaixImp.end() , FaixaImposto( 0, 100, 10) );
ListFaixImp.insert ( ListFaixImp.end() , FaixaImposto(101, 200, 15) );
ListFaixImp.insert ( ListFaixImp.end() , FaixaImposto(201, 0, 20) );
// Imprime Percentuais de imposto:
cout << endl << "Percentuais de imposto:" << endl;

for ( ListFaixImpIt = ListFaixImp.begin();
      ListFaixImpIt != ListFaixImp.end(); ++ListFaixImpIt )
{
    cout << ListFaixImpIt->m_Percent << " - ";
}
cout << endl;

// Calcula imposto para um valor:
float Imposto = 0;
float Valor = 150;
cout << endl << "Imposto para 150 reais:" << endl;
for (ListFaixImpIt = ListFaixImp.begin();
      ListFaixImpIt != ListFaixImp.end(); ++ListFaixImpIt )
{
    if ( Valor >= ListFaixImpIt->m_Min &&
        (Valor <= ListFaixImpIt->m_Max ||
         ListFaixImpIt->m_Max == 0))
    {
        Imposto = Valor * ListFaixImpIt->m_Percent / 100;
        break;
    }
}
cout << "Imposto = " << Imposto << endl;
Pausa();
}

```

## 9.4 • std::map

```
// ===== Usando map:
void TestaMap()
{
    cout << "MAP: Inicia teste:" << endl << endl;
    map_int_string MapIntStr;
    map_int_string::iterator MapIntStrIt;

    // Inserindo elementos no mapa:

    // Abaixo: "value_type" é um par "chave/valor":
    MapIntStr.insert(map_int_string::value_type(3,"Terceiro"));
    MapIntStr.insert(map_int_string::value_type(2,"Segundo"));
    MapIntStr.insert(map_int_string::value_type(1,"Primeiro"));
    // Procurando elementos pela chave:
    MapIntStrIt = MapIntStr.find(3);
    // procura pelo primeiro elemento do par(a chave)
    if ( MapIntStrIt != MapIntStr.end() )
        cout << MapIntStrIt->second << endl;
        // second é o segundo elemento do par (o valor associado);
        // desse modo, será impresso "Terceiro"
    MapIntStrIt = MapIntStr.find(2);
    if ( MapIntStrIt != MapIntStr.end() )
        cout << MapIntStrIt->second << endl;
        // será impresso "Segundo"

    MapIntStrIt = MapIntStr.find(1);
    if ( MapIntStrIt != MapIntStr.end() )
        cout << MapIntStrIt->second << endl << endl;
        // será impresso "Primeiro"

    // Percorrendo e imprimindo o map:
    for ( MapIntStrIt = MapIntStr.begin() ;
          MapIntStrIt != MapIntStr.end() ; MapIntStrIt++ )
    {
        cout << MapIntStrIt->first << endl; //imprime a chave;
        cout << MapIntStrIt->second << endl;
        //imprime o valor associado;
    }
    cout << endl;

    // eliminando elementos:
    MapIntStr.erase(2);

    // reimprimindo o map após a exclusão do elemento com chave 2:
    MapIntStrIt = MapIntStr.begin();
    while ( MapIntStrIt != MapIntStr.end())
    {
        cout << MapIntStrIt->second << endl;
        MapIntStrIt++;
    }
    cout << endl;

    // esvaziando todo o map:
    MapIntStr.clear();

    // reimprimindo o map (NADA será impresso agora):
    for ( MapIntStrIt = MapIntStr.begin() ;
          MapIntStrIt != MapIntStr.end() ; MapIntStrIt++ )
    {
```

```

        cout << MapIntStrIt->second << endl;
    }
    Pausa();
}

```

## 9.5 • std::vector < std::vector < > >

```

// ===== Usando um vetor de vetores:
void TestaVectorVector()
{
    cout << "VECTOR<VECTOR>: Inicia teste:" << endl << endl;

    int Linhas = 5;
    int Colunas = 3;
    vector<int> vec2Int;

    // Irá criar 5 vetores de int dentro do vetor de vetores;
    // sendo que cada vetor de int's terá 3 elementos:

    // primeiro cria um vetor com 3 elementos iniciados com zero:
    vector<int> item;
    item.assign(Colunas,0); // um vetor de 3 inteiros zerados.

    // agora, dimensiona o vetor de vetores com tamanho 5,
    // preenchendo esses 5 elementos com o vetor de 3 int's:
    vec2Int.assign(Linhas, item);

    // agora vamos preencher, sabendo-se
    // que o vetor é indexado a partir de ZERO (como uma matriz):
    vec2Int[0][0] = 11; // "linha" 1, "coluna" 1
    vec2Int[0][1] = 12; // "linha" 1, "coluna" 2
    vec2Int[0][2] = 13; // "linha" 1, "coluna" 3
    vec2Int[1][0] = 21; // "linha" 2, "coluna" 1
    vec2Int[1][1] = 22; // "linha" 2, "coluna" 2
    vec2Int[1][2] = 23; // "linha" 2, "coluna" 3
    // e etc...

    // agora, imprimir:
    for (int iLinha = 0; iLinha < vec2Int.size(); iLinha++)
    {
        for (int iColuna=0; iColuna < vec2Int[iLinha].size(); iColuna++)
        {
            cout.width(3);
            cout << vec2Int[iLinha][iColuna] << " - ";
        }
        cout << endl;
    }

    /* RESULTADO:

    VECTOR<VECTOR>: Inicia teste:

    11 - 12 - 13 -
    21 - 22 - 23 -
    0 - 0 - 0 -
    0 - 0 - 0 -
    0 - 0 - 0 -

```



```

        Todas as colunas de todas as linhas foram iniciadas com zero
        ("assign");
        Depois as duas primeiras linhas tiveram seus valores alterados
        manualmente.
        Assim, o resultado está correto.
    */
    Pausa();
}

void Pausa()
{
    cout << endl << " tecle <enter> para continuar..." << endl;
    cin.get(); // espera pela entrada no teclado e depois despreza;
    cout << endl << endl ;
}

```

## 9.6 • Laço *for* para *containers* (C++11)

O Laço *for* baseado em intervalos é usado para percorrer o conteúdo de qualquer classe *container* que suporte o conceito de intervalos e que possua os métodos *begin* e *end*.

```

#include <iostream>
#include <list>

using namespace std;

int main()
{
    list<int> li;
    li.push_back(1);
    li.push_back(2);
    li.push_back(3);
    li.push_back(4);
    li.push_back(5);

    /*
    Sintaxe anterior para percorrer containers
    for(list<int>::iterator it = li.begin(); it != li.end(); ++it)
        cout << *it << '\n';
    */

    //Sintaxe C++11
    for(auto it : li)
        cout << it << '\n';

    /*
    Ou com referencia
    for(auto &it : li)
        cout << it << '\n';
    */

    return 0;
}

```

## 9.7 • Outras classes containers

Além dos tipos mostrados anteriormente, a STL contém muitos outros tipos que podem ser usados de acordo com a necessidade de cada um. Alguns outros exemplos são:

### 9.7.1 • `initializer_list` (C++11)

Armazena uma coleção de valores que devem ser do mesmo tipo, assim como os *arrays*. Para armazenarmos valores nessa lista usamos as chaves `{}`. Geralmente usada para inicializar ou atribuir valores a uma classe *container*.

Para percorrermos um *initializer\_list* usamos *iterator* assim como qualquer outra classe *container*.

As **construtoras** de classes que recebem um *initializer\_list* como parâmetro tem **precedência** sobre as outras.

```
#include <iostream>
#include <initializer_list>

using namespace std;

class Teste
{
public:
    //Construtora que recebe dois ints e os imprime
    Teste(int a, int b)
    {
        cout << "\n***Teste(int a, int b)***\n";
        cout << a << '\n' << b << '\n';
    }

    //Construtora que recebe uma lista de initializer_list e imprime os itens
    Teste(initializer_list<int> l)
    {
        cout << "\n***Teste(initializer_list<int> l)***\n";
        //Percorre os elementos da lista e exibe na tela
        for(auto it = l.begin(); it < l.end(); it++)
            cout << *it << '\n';
    }
};

int main()
{
    Teste s1({1, 2}); //Chama Teste(initializer_list<int> l)
    Teste s2{10, 20}; //Chama Teste(initializer_list<int> l)
    Teste s3(100, 200); //Chama Teste(int a, int b)

    return 0;
}
```

O *template* de classe **`std::vector`** também passou a ter uma construtora que recebe uma *initializer\_list* como argumento:

```
#include <iostream>
```

```
#include <vector>

using namespace std;

int main()
{
    //Cria 5 elementos com o valor inicial de 10
    vector<int> v1(5, 10);

    //Cria 2 elementos (5 e 10)
    vector<int> v2{5, 10}; // não se engane: com parênteses criou 5 elementos; aqui só 2.

    for(auto it = v1.begin(); it < v1.end(); ++it)
        cout << *it << '\n';

    cout << "\n";

    for(auto it = v2.begin(); it < v2.end(); ++it)
        cout << *it << '\n';

    return 0;
}
```

## 9.7.2 • forward\_list (C++11)

Lista ligada unidirecional, só é possível inserir elementos a partir do início.

### 1. Exemplo

```
#include <iostream>
#include <forward_list>
#include <iterator>

using namespace std;

// Exibe conteúdo da forward_list
template<typename Tipo>
void imprimir(forward_list<Tipo> li)
{
    for(auto it: li)
        cout << it << '\n';
    cout << "\n\n";
}

int main()
{
    // Monta forward_list com 5 elementos
    forward_list<int> list = {3, 4, 5, 6, 7};
    imprimir(list);

    // Insere antes do primeiro elemento
    list.insert_after(list.before_begin(), 1);
    imprimir(list);

    // Insere depois do primeiro elemento
    list.insert_after(list.begin(), 2);
    imprimir(list);
}
```

```
// Insere antes do primeiro elemento
list.push_front(0);
imprimir(list);

//Erro, não pode inserir elementos no fim
//list.insert_after(list.end(), 8);

return 0;
}
```

## 2. Exemplo

```
#include <iostream>
#include <forward_list>
#include <iterator>
#include <algorithm>

using namespace std;

// Imprime conteúdo das listas
void imprimir(const string title, const forward_list<int>& l1,
              const forward_list<int>& l2)
{
    cout << title << '\n';
    cout << " list1: ";

    for(auto it: l1)
        cout << it << " ";
    cout << endl << " list2: ";

    for(auto it: l2)
        cout << it << " ";
    cout << endl;
}

int main()
{
    //Cria duas forward_list
    forward_list<int> list1 = {1, 2, 3, 4};
    forward_list<int> list2 = {66, 77, 88, 99};

    //Exibe as listas
    imprimir("Inicio:", list1, list2);

    //Insere 55 antes do primeiro elemento (66)
    list2.insert_after(list2.before_begin(), 55);

    //Insere 44 antes do primeiro elemento (55)
    list2.push_front(44);

    //Insere 11, 22, e 33 antes primeiro elemento (44)
    list2.insert_after(list2.before_begin(), {11,22,33});
    imprimir("\n\n5 novos elementos em list2:", list1, list2);

    //Insere todos os elementos da lista2 dentro da lista 1
    list1.insert_after(list1.before_begin(), list2.begin(), list2.end());
    imprimir ("\n\nInsere todos os elementos de list2 em list1:", list1, list2);

    //Deleta o segundo elemento da list2, ou seja, 22
    list2.erase_after(list2.begin());
    imprimir ("\n\nDeleta o segundo elemento da list2, ou seja, 22:", list1, list2);
}
```

```

//find irá apagar todos os elementos após 77
list2.erase_after(find(list2.begin(),list2.end(), 77), list2.end());
imprimir ("\n\nDeleta o elemento 99 da list2:", list1, list2);

//Reordena as posições
list1.sort();

//copia list1 dentro de list2
list2 = list1;

//Remove duplicações
list2.unique();
imprimir("\n\nsorted e unique:", list1, list2);

//Retira todos os elementos de list2 e insere em list1
list1.merge(list2);
imprimir ("\n\nmerged:", list1, list2);

return 0;
}

```

### 9.7.3 • unordered\_map (C++11)

**Containers associativos** que armazenam os elementos formados pela combinação de um valor e uma chave, e que permite a rápida recuperação de elementos individuais com base em suas chaves.

```

#include <iostream>
#include <unordered_map>

using namespace std;

int main()
{
    unordered_map<string, int> coll{ {"unordered_map_10", 10},
                                     {"unordered_map_20", 20} };

    //Imprime o elemento e a chave
    for (const auto &elem : coll)
        cout << elem.first << ": "
              << elem.second << '\n';

    return 0;
}

```

### 9.7.4 • unordered\_multimap (C++11)

**Containers associativos** que armazenam os elementos formados pela combinação de um valor e uma chave, assim como recipientes **unordered\_map**, mas permitindo que diferentes elementos tenham chaves equivalentes.

```

#include <iostream>
#include <unordered_map>

using namespace std;

```

```
int main()
{
    unordered_multimap<string, int> coll{ {"unordered_map_10", 10},
                                           {"unordered_map_10", 10} };

    //Imprime o elemento e a chave
    for(const auto &elem : coll)
        cout << elem.first << ": "
              << elem.second << '\n';

    return 0;
}
```

### 9.7.5 • unordered\_set (C++11)

**Containers** que armazenam elementos únicos em nenhuma ordem particular, e que permitem a rápida recuperação de elementos individuais com base em seu valor.

```
#include <iostream>
#include <unordered_set>

using namespace std;

int main()
{
    unordered_set<string> coll{"Numero: 111", "Quedinho", "Major", "Rua"};

    //Imprime o elemento
    for(auto &it : coll)
        cout << it << " ";

    cout << "\n\n";
    return 0;
}
```

### 9.7.6 • unordered\_multiset (C++11)

**Containers** que armazenam elementos sem nenhuma ordem particular, permitindo rápida recuperação de elementos individuais com base em seu valor, assim como os **containers unordered\_set**, mas permitindo que diferentes elementos tenham valores equivalentes.

```
#include <iostream>
#include <unordered_set>

using namespace std;

int main()
{
    unordered_multiset<string> coll{"Numero: 111", "Quedinho", "Major", "Rua",
                                     "Numero: 111", "Quedinho", "Major", "Rua"};

    //Imprime o elemento
    for (auto &it : coll)
        cout << it << " ";

    cout << "\n\n";
}
```

```
    return 0;
}
```

### 9.7.7 • array (C++11)

*Containers* de tamanho fixo que armazenam um número específico de elementos ordenados em sequência.

```
#include <iostream>
#include <array>
#include <algorithm>

using namespace std;

//Imprime todos os elementos
void imprime(array<int, 10> &a)
{
    for(auto &it : a)
        cout << it << '\n';
}

int main()
{
    //Cria um array de ints com 10 elementos
    array<int, 10> a = { 10, 20, 30, 40 };
    imprime(a);

    int total = accumulate(a.begin(), a.end(), 0);
    cout << "\n\nSoma de todos os elementos: " << total << '\n';

    return 0;
}
```

### 9.7.8 • tuple (C++11)

*Tuples* são objetos que podem armazenar elementos de tipos diferentes ao mesmo tempo.

```
#include <iostream>
#include <tuple>

using namespace std;

int main()
{
    //Cria uma tuple que armazena 5 valores string
    tuple<string, string, string, string> t("Herik",
                                           "RG:00.000.000-00",
                                           "CPF:000.000.000-00",
                                           "11/08/1991");

    //O Template de função get nos permite acessar cada elemento da tupla pela ordem:
    cout << get<0>(t) << ", ";
    cout << get<1>(t) << ", ";
    cout << get<2>(t) << ", ";
    cout << get<3>(t) << "\n\n";
}
```

```
    return 0;  
}
```



---

## • Capítulo 10

### ▪ Classes e objetos: herança

---

10.1 • Regras básicas.....	290
10.1.1 ▪ Restrição de acesso <i>protected</i> .....	290
10.1.2 ▪ Modos de derivação.....	290
10.1.2.1 ▪ Derivação pública ( : public).....	290
10.1.2.2 ▪ Derivação privada ( : private).....	290
10.1.2.3 ▪ Derivação protegida ( : protected).....	290
10.2 • Quando usar herança.....	291
10.3 • Construtoras e destrutoras na derivada e na base.....	292
10.3.1 ▪ Criação de um objeto de uma classe derivada.....	292
10.3.2 ▪ Destruição de um objeto de uma classe derivada.....	292
10.3.3 ▪ Como a construtora derivada pode chamar uma construtora base.....	292
10.3.4 ▪ Herdando construtoras (C++11).....	293
10.4 • Herança Múltipla.....	295
10.4.1 ▪ Herança múltipla e herança virtual:.....	296

## 10.1 • Regras básicas

Podemos **derivar uma classe de outra, reaproveitando** assim todo o código escrito (membros de dados e funções-membro) e acrescentando novas características à classe derivada.

A regra de derivação em si mesma é muito simples, bastando acrescentar o operador de derivação (`:`) imediatamente após uma classe ser nomeada.

Esse operador deve ser seguido de uma das restrições de acesso previstas por C++, indicando um modo de derivação, o qual permite elevar o nível de privacidade dos membros herdados da base na nova classe derivada (veremos esses modos logo abaixo):

Exemplo de derivação (classe "B", derivada de "A"):

Classe base:	Classe derivada:
class A	Class B : public A // "B" é uma derivada de "A"
{ } ;	{ } ;

### 10.1.1 ▪ Restrição de acesso *protected*.

Além das seções "**public**" e "**private**", devido a herança, existe uma terceira restrição de acesso a membros: a seção "**protected**" (protegida), de tal modo que:

- os membros "**public**" da classe-base serão públicos na classe derivada;
- os membros "**private**" da classe-base, **não podem ser acessados** pelas funções-membro da classe derivada (serão sempre membros "**private**" da base);
- já os membros declarados como "**protected**" em uma classe podem ser **acessados tanto nessa classe, como em suas amigas e, ainda, nas funções de suas classes derivadas** (e não podem ser acessados em qualquer outro escopo).

Contudo, em uma derivada, esses níveis de restrição de acesso podem ser forçados para um nível maior de privacidade, como veremos agora, nos *modos de derivação*.

### 10.1.2 ▪ Modos de derivação.

Temos três modos de derivar uma classe da outra:

#### 10.1.2.1 ▪ Derivação pública ( : public).

- neste caso os membros **públicos e protegidos** da classe base serão respectivamente públicos e protegidos na classe derivada, mantendo assim seu status;
  - os membros "**private**" da classe-base **continuam naturalmente restritos à classe-base**.

#### 10.1.2.2 ▪ Derivação privada ( : private).

- os membros **públicos e protegidos** da classe-base **tornam-se "private" na classe derivada**; no mundo externo, só podem ser acessados assim por funções-membro da classe derivada;
- os membros "**private**" da classe-base **continuam naturalmente restritos à classe-base**.

#### 10.1.2.3 ▪ Derivação protegida ( : protected).

- os membros **públicos e protegidos** da classe-base **tornam-se "protected" na classe derivada**; só podem ser acessados assim por funções-membro de ambas as classes,

por suas respectivas amigas, ou por funções de uma nova classe que escolha como classe-base a atual derivada.

- os membros “**private**” da classe-base **continuam naturalmente restritos à classe-base**.

## 10.2 • Quando usar herança



Em C++ herança é uma forma eficaz e segura de reaproveitamento de código. Deve ser usada sempre que houver uma hereditariedade real entre os componentes envolvidos.

Faz sentido, por exemplo, derivar a classe “BotãoComBitmap” da classe “Botão”;



Mas não faria muito sentido derivar a classe “Menu” da Classe “Data”, mesmo que eventualmente “Menu” precise usar “Data” (por exemplo, para exibir uma data na tela).

Neste caso, seria melhor mantê-las independentes.

E bastaria que um dos membros de dados de “Menu” fosse um objeto da Classe “Data” - ou um ponteiro para um objeto “Data”.

// a **classe-base** “cadastro”

```
class Cadastro
{
    private:
        int iStatusInterno;

    protected:
        char Nome[ 30 ];
        char CGC[20];
        char Endereco[40];
        char Cidade[32];
        char CEP[8];

    public:
        int iQualquerCoisa; // sem validação
        ..... FUNÇÕES .....
};
```

// a classe “clientes” é uma **classe-derivada** de “cadastro”:

```
class Clientes : public Cadastro
{
    ..... ACRESCENTA SEUS PRÓPRIOS MEMBROS .....
}
```

// a classe “fornecedores” também é uma **classe-derivada** de “cadastro”:

```
class Fornecedores: public Cadastro
{
    ..... ACRESCENTA SEUS PRÓPRIOS MEMBROS .....
}
```



Tanto fornecedores como clientes herdam as características de cadastro (que devem ser exatamente os dados cadastrais básicos de ambas).

## 10.3 • Construtoras e destrutoras na derivada e na base

### 10.3.1 • Criação de um objeto de uma classe derivada.

Quando um objeto de uma classe **derivada** é **criado**, o seguinte fluxo de processamento é seguido:

- 1) É chamada a **construtora** da classe **derivada**.
- 2) **Antes** que suas instruções (entre as chaves da função) sejam executadas é chamada a **construtora** da classe **base**.
- 3) A **construtora** da classe **base** é então executada. Isso significa que ela é executada antes que a construtora derivada seja executada (o que faz sentido, pois é preciso que os membros da base sejam inicializados antes que a derivada tente utilizá-los).
- 4) Finalmente, **após o retorno** da construtora base, será executada a **construtora derivada**.

### 10.3.2 • Destruição de um objeto de uma classe derivada.

Quando um objeto é **liberado**, o seguinte fluxo de processamento é seguido:

- 1) É chamada e executada a **destrutora** da classe **derivada**
- 2) Imediatamente **após o seu retorno**, é chamada a **destrutora** da classe **base**.

Desse modo, a lógica dos objetos (disparos automáticos de funções para criação e destruição) é simples e naturalmente desdobrada para adequar-se ao conceito de herança.

### 10.3.3 • Como a construtora derivada pode chamar uma construtora base.

Vimos que a construtora derivada **sempre**, **antes** de ser efetivamente executada, chama a construtora base.

E se uma classe-base tiver mais do que uma construtora. Será chamada, *por default*, a construtora que não recebe parâmetros.

Se queremos modificar isso devemos, explicitamente, chamar uma determinada construtora, passando os parâmetros.

Fazemos isso usando o **operador de inicialização de membro**. Vejamos um exemplo.

```
class Base
{
    private :
        int m_iCodigo;
        .....
    public:
        Base ( ) ; // construtora
        Base ( int Cod ) ; // construtora com parâmetros
        .....
};
```

```

Base::Base ( ) // construtora-base
{
    m_iCodigo = 0;
}

Base::Base ( int iCod ) // construtora-base com parâmetros
{
    m_iCodigo = iCod ;
}

class Derivada : public Base // derivada
{
    .....
public:
    Derivada ( ) ;
    Derivada ( int iCod ) ;
}

Derivada::Derivada ( ) // construtora-derivada
    : Base ( ) // chama a construtora-base que não recebe parâmetros,
                // usando o operador de inicialização de membros,
                // antes do corpo da função.
                // Neste caso isto é desnecessário, pois é exatamente
                // isso que será feito por default,
                // já que está chamando a construtora default da base
{
    .....
}

/* Contudo, a derivada poderia usar isso para forçar um valor inicial:
Derivada::Derivada ( ) // construtora-derivada
    : Base ( 1 ) // chama a construtora-base que recebe parâmetros,
                // passando um valor inicial (constante já que esta
                // construtora derivada não tem parâmetros

    {
        .....
    }

*/

// Agora a construtora derivada que recebe parâmetros, poderá
// repassar os parâmetros recebidos para a construtora-base:
Derivada::Derivada ( int iCod ) // construtora-derivada com parâmetros
    : Base ( iCod )
        // chama a construtora-base que recebe parâmetros,
        // repassando o parâmetro recebido.

    {
        .....
    }

```

### 10.3.4 • Herdando construtoras (C++11)

A partir do padrão C++11, é possível que uma classe derivada herde uma construtora da classe base.

## 1. Exemplo:

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

class Base
{
public:
    Base() = default;

    Base(int x)
    {
        cout << "Base: " << x << "\n";
    }

    Base(const char *pCh)
    {
        cout << "Base: " << pCh << "\n";
    }
};

class Derivada : public Base
{
public:
    using Base::Base;
    Derivada()
    {
        cout << "Derivada\n";
    }
};

int main()
{
    Derivada derivada_1;
    Derivada derivada_2(10);
    Derivada derivada_3("AGIT");

    return 0;
}
```

Como se pode ver, a classe Derivada não implementou o construtor que recebe um int como parâmetro e nem o construtor que recebe um const char \*, ela simplesmente herdou esses construtores da classe Base, porém ao instanciar objetos da Derivada, percebe-se que nenhum construtor da Derivada foi executado e se o construtor inicializasse membros de dados isso poderia ser um problema, pois esses membros não seriam inicializados. Para contornar esse problema devemos utilizar a inicialização de membros in-class.

## 2. Exemplo:

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

class Base
{
public:
    Base() = default;
```

```

Base(int x)
{
    cout << "Base: " << x << '\n';
}

Base(const char *pCh)
{
    cout << "Base: " << pCh << '\n';
}
};

class Derivada : public Base
{
public:
    using Base::Base;

    int inicializacaoInClass = 88888;

    int inicializacaoConstrutora;

    Derivada() : inicializacaoConstrutora(88888)
    {
        cout << "Derivada\n";
    }
};

int main()
{
    Derivada derivada_1;
    Derivada derivada_2(10);
    Derivada derivada_3("AGIT");

    cout << "derivada_1.inicializacaoInClass: " << derivada_1.inicializacaoInClass << '\n';
    cout << "derivada_2.inicializacaoInClass: " << derivada_2.inicializacaoInClass << '\n';
    cout << "derivada_3.inicializacaoInClass: " << derivada_3.inicializacaoInClass << '\n';

    cout << "derivada_1.inicializacaoConstrutora: " << derivada_1.inicializacaoConstrutora <<
'\n';
    cout << "derivada_2.inicializacaoConstrutora: " << derivada_2.inicializacaoConstrutora <<
'\n';
    cout << "derivada_3.inicializacaoConstrutora: " << derivada_3.inicializacaoConstrutora <<
'\n';

    return 0;
}

```

## 10.4 • Herança Múltipla



**Uma mesma classe pode ser herdeira de várias classes.**

```

class Vencimentos
{
public:
    float ValorVencs;
};

```

```

class Descontos
{
    public:
        float ValorDescs;
};

class SalarioFinal : public Vencimentos , public Descontos
{
    public:
        float ValorFinal ;
        float Calcula ( )
        {
            ValorFinal = ValorVencs - ValorDescs;
            return ValorFinal ;
        }
};

int main( )
{
    SalarioFinal MeuSalario;
    MeuSalario.ValorVencs = 10;
    MeuSalario.ValorDescs = 5 ;
    cout << MeuSalario.Calcula( ) << endl ; //IMPRIME 5;
}

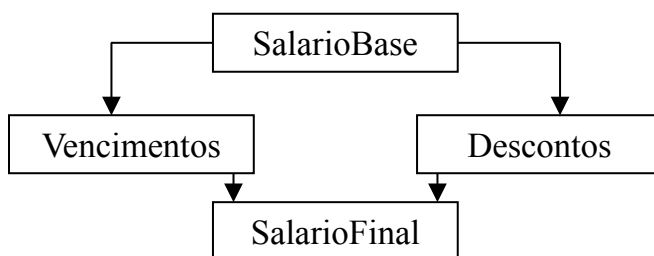
```

- A Classe **Salario Final** é herdeira de **Vencimentos** e **Descontos**. Isso significa que um objeto **SalarioFinal** comporta todos os membros de **Vencimentos** e de **Descontos** e também os seus próprios membros.

### 10.4.1 • Herança múltipla e herança virtual:



Se a regra acima é verdadeira o que aconteceria se **Vencimentos** e **Descontos** tivessem sua própria classe-base (da qual seriam derivadas)?



- O diagrama aparentemente poderia ser expresso assim:

```

class SalarioBase
{ ..... };
class Vencimentos : public SalarioBase
{ ..... };
class Descontos : public SalarioBase
{ ..... };
class SalarioFinal : public Vencimentos , public Descontos
{.....};

```

- Mas, no caso acima, estaríamos com a seguinte situação:
- Ao construirmos um objeto "MeuSalario" da classe **SalarioFinal**, é consultado o molde dessa classe; ocorre então que:

1) Ele informa que é derivado de **Vencimentos**;



- o molde de Vencimentos é então consultado;
- e este informa que é derivado de SalarioBase;
  - os membros de SalarioBase serão então considerados para a criação do objeto;
  - os membros de Vencimentos serão agora considerados na construção do objeto.

## 2) E SalarioFinal informa que também é derivado de Descontos;

- o molde de Descontos é então consultado;
- e este informa que é derivado de SalarioBase;
  - os membros de **SalarioBase** serão **novamente considerados** na criação do objeto;
  - os membros de Desconto serão normalmente considerados;
- **3** - Finalmente, os próprios membros de SalarioFinal serão considerados.



Desse modo teremos a **duplicação** dos membros do primeiro ancestral da hierarquia (a classe SalarioBase, no exemplo).

Isso pode também ficar muito confuso: seremos obrigados a usar a resolução de escopo para acessarr os membros de "SalarioBase" herdados via "Vencimentos" e via "Descontos".

- A ambiguidade será resolvida se especificarmos que Vencimento e Desconto são **virtualmente** herdeiros de SalarioBase (e **não necessariamente** herdeiros).

É o mesmo que dizer:



"Serão herdeiros se, e somente se, tal herança ainda não houver sido realizada".

- Para implementar a **herança virtual** deveríamos então reescrever nossas diretrizes de derivação do seguinte modo:

```
class SalarioBase
{ ..... };

class Vencimentos : virtual public SalarioBase
{ ..... };

class Descontos      : virtual public SalarioBase
{ ..... };

class SalarioFinal   : public Vencimentos , public Descontos
{ ..... };
```

- Desse modo SalarioBase seria herdado **apenas uma vez** em objetos da classe Salario-Final.
- E seria herdado normalmente em objetos da classe vencimentos ou objetos da classe descontos.



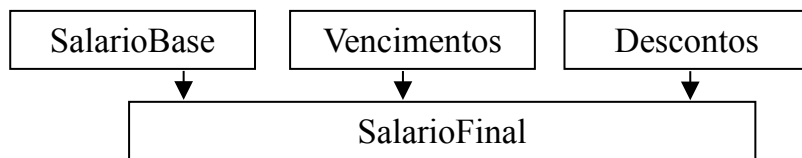
Quando temos a necessidade de herança virtual (classes sendo herdadas por vários caminhos), é preciso voltar **à análise do caso de uso**.

Talvez faça sentido.

Mas em muitas ocasiões revelam um erro de análise.  
O exemplo acima poderia ser resovido de outro modo.

**Afinal, "Vencimentos" e "Descontos", no mundo real, precisam derivar de "SalarioBase"?**

**Para este caso, uma solução mais apropriada provavelmente seria:**



```

class SalarioBase { ..... };
class Vencimentos { ..... };
class Descontos { ..... };
class SalarioFinal : public SalarioBase ,
                    public Vencimentos , public Descontos
{ ..... };
    
```

---

## • Capítulo 11

### ▀ Classes e objetos: Polimorfismo

---

11.1 • Sobrecarga.....	300
11.1.1.1 ▀ Sobrecarga de funções.....	300
11.1.1.2 ▀ Sobrecarga de operadores.....	300
11.2 • Redefinição de funções nas classes derivadas.....	302
11.3 • Funções virtuais (polimorfismo estrito).....	304
11.3.1 ▀ Polimorfismo em <i>frameworks</i> : ponteiros para função e funções virtuais.....	305
11.3.2 ▀ Declarar e implementar funções virtuais.....	306
11.3.3 ▀ Final e override (C++11).....	308
11.3.3.1 ▀ Final (C++11).....	308
11.3.3.2 ▀ Override (C++11).....	309
11.3.4 ▀ Funções virtuais como respostas a eventos.....	309
11.3.4.1 ▀ Funções virtuais puras e classes abstratas.....	312
11.3.4.2 ▀ Exercício: a classe <i>RelatPadrao</i> .....	312
11.3.4.2.a ▀ Solução: declaração, implementação e uso da classe <i>RelatPadrao</i> .	313
11.3.5 ▀ Precauções ao usar funções virtuais.....	314
11.3.5.1.a ▀ 1 - Nunca preencher um objeto com métodos de baixo nível.....	314
11.3.5.1.b ▀ 2 - Nunca esquecer como a <i>vtable</i> é preenchida.....	314
11.4 • Questões para revisão.....	315



Rigorosamente falando polimorfismo refere-se a **funções virtuais**.

**Diz-se que uma classe é polimórfica quando contém pelo menos uma função virtual.**

Mas há quem diga que, por fidelidade à palavra em sentido amplo (*polimorfismo = alguma coisa que assume múltiplas formas*), tudo aquilo que implicar em *múltiplas formas de uso de um recurso* deveria ser considerado no capítulo polimorfismo.

Sem entrar no mérito, iremos aqui incluir tais recursos.

## 11.1 • Sobrecarga.

### 11.1.1.1 • Sobrecarga de funções.



Observe que a **sobrecarga** (que vimos no capítulo **5**, página **131**) é válida para qualquer função, membra de classe ou não.

Observe também que a sobrecarga poderá ser especialmente útil em funções **construtoras** de classe, já que a sobrecarga permitirá diversas versões de modo a passar parâmetros e inicializar alguns membros de dados com valores diferenciados.

#### *Exemplo de sobrecarga com construtoras:*

```
class Data
{
    private:
        char Dia;
        char Mes;
        short Ano ;

    public:
        Data ( ) ; /* Não recebe parâmetros,
                    logo só poderá inicializar os membros com valores
                    default */

        Data ( char D, char M, short A ) ; /* poderá inicializar os membros
                                            Dia (a partir do parâmetro D),
                                            Mes (a partir do parâmetro M),
                                            Ano (a partir do parâmetro A) */

        Data ( short N ) ; /* só poderá inicializar um único membro
                            (a partir do parâmetro N);
                            os demais só poderão ser inicializados a partir de valores default
                            */
};
```

### 11.1.1.2 • Sobrecarga de operadores

Já vimos que em C++ podemos escrever nossos próprios operadores, como membros de uma classe ou como funções globais. Operadores são um elemento de **programação genérica** (capítulo **8**, página **264**)

- Na realidade, **um operador é uma função** com algumas características específicas.

- É escrito do mesmo modo que escrevemos uma função –só que não podemos escolher livremente o nome da função, devendo empregar um símbolo ou nome de operador já existente na linguagem.
- Para isto utilizamos o especificador **operator**, seguido de um **símbolo de operador** ( =, +, -, etc) já admitido pela linguagem.


**Exemplo:**

```
class Qualquer
{
    private:
        int Valor ;
    public:
        int operator = (int Param) ;
};
int Qualquer::operator = (int Param )
{
    Valor = Param;
    return Valor;
}

int main( )
{
    Qualquer Var;
    Var = 2 ;
}
```

- Uma função operadora redefine ou cria, para um determinado tipo de dados, um operador já existente na linguagem; então ela herda a regra de precedência do operador já existente na linguagem.
- Os parâmetros da função operadora são os **operandos** admitidos para cada operador; por isso existirá uma restrição quanto ao número de parâmetros (normalmente apenas um parâmetro explícito; e no máximo dois).
- Uma vantagem em usar operadores é que a escrita se torna mais legível.
- Mas uma outra vantagem, ainda **mais significativa**, é que os operadores permitem que classes diferentes **se comuniquem, contribuindo fortemente para programação genérica**.
- Considere, sobre isto, o seguinte exemplo (veremos em detalhes nos capítulos sobre *templates* e *STL*).

```
.....
list < Data > dtLst; // uma lista de objetos do tipo 'Data'
.....
dtLst.sort(); //classifica a lista de datas
```

 O uso do método "sort", aqui, só foi possível porque a **class Data** dispõe da função **operator <(...)** (operador "menor que"), o qual é usado pelo método sort, para classificar a lista.

## 11.2 • Redefinição de funções nas classes derivadas.

- Uma classe derivada pode redefinir funções de suas ancestrais.
- Isto significa que:
- uma classe base, por exemplo a classe "Janela", pode ter uma função chamada "Exibe";
- uma derivada, por exemplo a classe "JanelaComMensagem", pode também ter uma função chamada "Exibe"
- Objetos da classe "Janela" acessarão a sua própria função "Exibe".
- Objetos da classe "JanelaComMensagem" acessarão a função "Exibe" **desta classe** e não a função da classe-base (pois a função da derivada estará escondendo a ancestral).
- Contudo, dentro das funções da classe derivada, a função-base **pode ser chamada diretamente** através do operador de resolução de escopo, permitindo assim, inclusive, o reaproveitamento do código já escrito nas funções da classe base.

*Exemplo:*

```
class Janela    // classe base
{
    public:
        void Exibe( )
        {
            DesenhaBorda ( );
            ImprimeTitulo ( );
        }
        .....
};

class JanelaComMensagem : public Janela // derivada
{
    public:
        void Exibe( )           // redefine função da classe ancestral
        {
            Janela::Exibe( ) ;   // chama diretamente a função ancestral
            ImprimeMensagens ( ) ; // acrescenta código específico
        }
        .....
};

int main( )
{
    Janela objJanela ;
    objJanela.Exibe ( ) ; // será usada a função Exibe() da classe-base

    JanelaComMensagem objJanMsg ;
    objJanMsg.Exibe( ) ;   // será usada a função Exibe() da classe-derivada

    return 0;
}
```

Vimos então que a redefinição de funções de classes-base em uma classe derivada permite ocultar as funções-base.

Este recurso irá permitir que uma função possa ser simplesmente **substituída** em classes derivadas.

Vamos usar mais um exemplo de redefinição de funções para depois melhor entendermos o uso de funções virtuais.

**Um exemplo.**

Vamos criar uma classe para servir como base para a construção de tabelas de referência. Depois criaremos uma derivada para implementar um tipo de tabela mais específico.

### 1) classe base:

- Ela cuida de apenas dois campos, sempre usados em tabelas de referência: os campos "código" e "Descrição".
- Para código, a classe implementa a regra de validação mais comum: o código deve estar entre um mínimo e um máximo. Então a classe base fornece uma função para que sejam informados os valores mínimo e máximo. Contudo, ela exige que esta função seja chamada apenas uma vez para cada objeto criado (de tal modo que novas chamadas a essa função serão simplesmente desprezadas). Com isso, o mínimo e o máximo definem um único comportamento para o campo "código", durante todo o tempo de vida de cada objeto. Já o campo "Descrição" será considerado válido se preenchido com uma quantidade mínima de caracteres estipulada pela classe.
- Além disso, a classe base implementa um método para impressão de código e descrição: se o código estiver correto e se a descrição estiver preenchida com a quantidade mínima de caracteres, os valores serão impressos; do contrário, será impressa uma mensagem de erro.

### 2) classe derivada:

- Agora, escreveremos uma classe derivada que poderá acrescentar outros campos; neste caso, acrescentaremos o campo "Tipo" (que servirá para identificar se o par código/descrição se refere a uma receita ou a uma despesa).
- Se criarmos objetos a partir da classe derivada e em seguida usarmos o método "Imprimir", serão impressos apenas os campos "código" e "descrição" –pois, até aqui, a única função de impressão existente é a função da classe-base que conhece apenas esses dois campos.
- Esse método não é assim realmente útil, pois deveria ser impresso também o campo "Tipo". Para resolver o problema, a classe derivada deve implementar o **seu próprio método** "Imprimir".
- Ao incluir esse novo método, que terá o **mesmo nome** do método correspondente na classe base, o que acontece é que, para objetos criados a partir da classe-derivada, a função da classe-base é "escondida", isto é, é **anulada** pela função da derivada. Assim, para estes objetos, será sempre chamada a função da classe derivada.
- Dentro da função-derivada, ela pode contudo (caso seja conveniente), **reaproveitar** aquilo que já é feito pela função-base, chamando explicitamente essa função através do operador de resolução de escopo.
- Por exemplo:

```
// redefinição da função de mesmo protótipo na classe base:
bool derivada::Imprimir( bool bFimLinha )
{
    if ( base::Imprimir( false ) ) // reaproveita função da classe base
        // (sem fim de linha)
    {
        // a qual imprime código/descrição, ou uma mensagem de erro

        // agora acrescenta a impressão do TIPO :
        switch( m_iTipo )
        {
            case RECEITA:
                cout << " - RECEITA "; break;

            case DESPESA:
                cout << " - DESPESA "; break;

            default:
                cout << " - INVALIDO "; break;
        }
    }
}
```

```

    }
    if ( bFimLinha )
        cout << endl;
    return true;
}

```

A observação mais importante que podemos fazer é que a redefinição de funções de classes-base anula tais funções, permitindo assim a sua substituição de modo eficiente e sem que as classes-base precisem ser alteradas.

Isto é feito de modo muito simples:

- se uma variável é **declarada** a partir de uma classe **base**, e se uma função é **chamada a partir dessa variável**, será sempre utilizada a função membro dessa classe;
- se uma variável é **declarada** a partir de uma classe **derivada**, e se uma função é **chamada a partir dessa variável**, será sempre utilizada a função membro dessa classe, **anulando** todas as funções da classe base que tenham esse mesmo nome.

Quem determina portanto qual função deve ser chamada é o **tipo da variável** que dispara a chamada de função (isto é, a variável que terá seu endereço passado no parâmetro “**this**”).

As funções estão sendo chamadas **diretamente** por variáveis; a variável tem um **tipo**; o tipo tem funções próprias; simplesmente será usada a função correspondente ao tipo.

Assim, quando fazemos:

```

    base oB;
    oB.Imprime( );
OU
    derivada oD;
    oD.Imprime( );

```

não há dúvidas sobre qual função deve ser usada em cada caso, já que o tipo da variável está claro, e, se **cada classe** possuir a sua **própria** função “Imprime”, é **essa** função própria que deverá ser chamada.

Mas nem sempre temos uma situação assim tão clara e direta. E é aqui que entra um outro elemento da linguagem: as funções **virtuais**.

## 11.3 • Funções virtuais (polimorfismo estrito)

Esse é o elemento **essencial de polimorfismo**. Por isso mesmo dizemos que **uma classe é “polimórfica”, quando ela dispõe de funções virtuais**.

---

 Inicialmente devemos destacar que **polimorfismo** é um princípio **muito antigo** usado na indústria desde a segunda onda da **Revolução Industrial**. Contudo, em programação, **muitas vezes deixamos de tirar proveito** desse princípio tão antigo.

---



---

 **Um exemplo típico:**

---



### 11.3.1 • Polimorfismo em *frameworks*: ponteiros para função e funções virtuais

Um *framework* tipicamente é um “motor” que oferece um serviço **incompleto**. Por exemplo, um “loop” com certas características: ele saber “girar” mas não sabe exatamente o que fazer com esse “giro” em termos de uma aplicação final.

A utilidade de um “*framework*” está exatamente em implementar uma estrutura genérica de aplicação que fornece a funcionalidade previsível e comum a uma família de problemas.

Por isso, nesses casos, o melhor é **isolar o “motor” da sua “aplicação”**, pois assim permitimos que ele seja **reaproveitado** para diversas situações, para muitas “aplicações”. E esse comportamento “multi-uso” é uma das manifestações de **polimorfismo**.

É o caso de uma “furadeira elétrica” que na verdade faz bem mais do que apenas furos:



Isso é exatamente aquilo que podemos chamar de **polimorfismo**: uma mesma ferramenta básica que pode trabalhar de múltiplas formas. É flexível e a sua **flexibilidade** reside justamente no fato de que essa é uma **ferramenta incompleta**. Assim, ela só será útil se adquirirmos também a broca, a lixa, e o que mais for necessário para os nossos usos.

Mas se, por exemplo, ela tivesse uma broca soldada no bico, então ela se tornaria rígida, e não polimórfica: ela estaria **completa**, sim, mas nos forneceria apenas uma e **soamente uma funcionalidade**...

E ficaria bem mais caro adquirir (e também manter) diversos motores todos eles “completos”, todos amarrados a uma utilidade fixa.

E não é muito diferente tratando-se de *softwares*.

Também para programas de computador, se há alguma família de problemas em que um “motor” poderia servir para muitas situações, então a solução é também **separar o “motor” das suas “aplicações”**.

O engate (o “mandril”) nesse caso será um **ponteiro para função** acoplado ao “motor”.

Então, nas aplicações específicas, só precisaremos “**encaixar**” nesse “**bico do motor**” o **endereço de funções** específicas para cada caso: função “*Broca*”, função “*Serra*”, função “*Lixa*”, função “*Escova*”, etc, de acordo com as necessidades de cada situação.

E há duas maneiras de fazer isso:

- trabalhando **diretamente** com os endereços de funções através de **ponteiros para função**;
- trabalhando **indiretamente** com esses endereços através de **funções virtuais**;

Do ponto de vista da orientação a objetos, a melhor escolha seria o uso de **funções virtuais**, pois ele ocorre de modo estruturado, seguindo a lógica e o tempo de vida dos objetos.

Contudo ponteiros para função tem um importante uso em bibliotecas C++, como é o caso do mecanismo de signals/slots criado por **Qt** (e seguido depois por outras bibliotecas com a **libsigC++** e a **boost**)

### 11.3.2 • Declarar e implementar funções virtuais.

Frequentemente precisamos criar classes para implementar tarefas genéricas. E, em muitos desses casos, queremos que a classe genérica contenha uma determinada lógica, comum a problemas daquela natureza.

Isto é: a classe deve garantir que um conjunto de operações seja executado sempre com uma determinada sequência, o que pode implicar na necessidade de disparar chamadas de funções nos momentos adequados.

Contudo, algumas dessas funções podem estar **incompletas** (pois a classe base, para ser útil, deve cuidar exclusivamente da parte genérica), realizando assim apenas uma parte do trabalho necessário.

Pior do que isso: em muitos casos a classe base não terá condições de oferecer nem mesmo uma função incompleta. Pois ela sabe **apenas** que, em determinado momento, deve ser executada uma determinada função; ela sabe **qual o papel** que deve ser desempenhado por essa função (o que significa que ela conhece o **tipo** da função e o objetivo geral do trabalho que deve ser executado por ela) mas **não** sabe **como** a função deve ser implementada (pois isto depende da situação específica).

Nessa última situação, a classe base estará chamando uma função que na realidade **não está implementada** ainda (isto é: não está implementada no escopo dessa classe). Em consequência, **obrigatoriamente** deverá existir uma **classe derivada** que implemente a função (pois simplesmente não é possível chamar uma função que não exista em algum lugar).

Observe a **diferença** entre esta situação e a anterior. Na situação anterior, a classe base contém uma determinada função e uma classe derivada contém uma função com o mesmo nome. Se criarmos um objeto a partir da classe base, para esse objeto será usada a função-base. E, se um objeto for criado a partir da derivada, para esse objeto será usada a função-derivada. Isto porque estamos falando de uma função que estará sendo chamada **diretamente** pelo objeto:

[ objetoBase.Imprime( ); **ou**: objetoDerivado.Imprime( ); ].

Já na segunda situação, teremos uma função da classe-base (que simplesmente desconhece a existência de derivadas) chamando uma **outra função da própria classe-base**. Contudo, na maioria dos casos, queremos que seja executada uma função de alguma classe **derivada** (para que seja realizado um trabalho específico, adequado a uma determinada situação). E aqui temos as duas possibilidades assinaladas acima:

- Uma função de uma determinada classe base ( "Funcao\_1" ) chama outra função da mesma classe ( "Funcao\_2" ). "Funcao\_2" é implementada na classe base de modo genérico e é **incompleta** para a maioria das aplicações. Então escrevemos uma derivada que implementa a sua própria "Funcao\_2", para que seja executado o trabalho específico (e completo) necessário nesse caso. O código ficaria assim:

```
class base
{
    public:
        .....
        void Funcao_1( )
        {
            .....
            Funcao_2(); // exatamente neste ponto Funcao_2( )
                        // deve sempre ser executada.
            .....
        }
        void Funcao_2()
        {
            AbrirArquivoLog(); // Funcao_2 sabe que o arquivo de ocorrências
                               // deve ser aberto
            GravarRegistroCabecalho(); // e que um registro "header"
                                       // deve ser gravado;
                                       // mas não sabe o que deverá ser gravado a seguir.
        }
        ~base() // destrutora
        {
            FecharArquivoLog();
        }
        void AbrirArquivoLog();
        void FecharArquivoLog();
        void GravarRegistroCabecalho();
};

class derivada : public base
{
    void Funcao_2()
    {
        base::Funcao_2( );
    }
};
```

- Observe-se que nesse caso a função que gostaríamos de redefinir (*funcao\_2*) é chamada dentro de uma função **da própria classe base**.
- E a classe base desconhece a existência de derivadas. Como fazer então para que essa função possa ser redefinida e **substituída** por uma função de uma classe derivada?
- Se a declararmos como **virtual**, ela **será chamada a partir de um ponteiro para função** e não rigidamente a partir do seu nome.
- Assim quando declaramos uma função como virtual, o compilador irá criar um **membro de dados oculto e obrigatório**.
- Esse membro de dados é uma **matriz de ponteiros para função** denominada **vtable**.

- E para cada função virtual será adicionado um elemento nessa matriz de ponteiros, de tal forma que a matriz possa armazenar os endereços de todas as funções virtuais da classe.
- Antes que a construtora-base seja executada, a matriz é preenchida com os endereços das funções da classe base.
- Mas, após o retorno da construtora-base e antes que seja executada a construtora-derivada, os endereços são alterados: a matriz será então preenchida com os endereços das funções-derivadas (somente para aquelas que foram redefinidas).

### 11.3.3 ▪ Final e override (C++11)

As palavras final e override são palavras reservadas dependendo do contexto onde aparecem.

#### 11.3.3.1 ▪ Final (C++11)

Usada para uma classe ou para uma função virtual. Se usado para uma classe, indica que esta não pode ser herdada. Já para uma função virtual indica que ela não pode ser redefinida na classe derivada.

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

class Base: final {
public:
    int a;
};

/* Gera erro de compilação, pois base não pode ser herdada
class Derivada : public Base
{
    public: int b;
};
*/

class VirtualFinal
{
public:
    virtual void f() final{}
};

/*Gera erro de compilação, pois f() não pode ser redefinida

class DerivadaVirtualFinal: public VirtualFinal
{
    public: void f(){}
};
*/

int main()
{
    return 0;
}
```

### 11.3.3.2 • Override (C++11)

Usado para informar que a função está sobrecarregando uma virtual da classe base.

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

```
struct Base
{
    virtual void f()
    {
        cout << "F Base.\n";
    }
public:
    void executaF()
    {
        f();
    }
};

struct Derivada : Base
{
    //Indica que estamos reescrevendo a virtual da base
    virtual void f() override
    {
        cout << "Derivada override.\n";
    }
};

int main()
{
    Derivada de;
    de.executaF();

    return 0;
}
```

### 11.3.4 • Funções virtuais como respostas a eventos.

Já vimos isto quando falamos de ponteiros para funções. E funções virtuais têm a mesma flexibilidade de um ponteiro para função, só que utilizando a **lógica própria aos objetos**.

*Vejamos um exemplo.*

Iremos implementar a classe “Despertador”. A classe “Despertador” sabe que, de tempos em tempos (a cada cinco segundos, por exemplo), deve disparar uma chamada de função para que algum trabalho seja feito. Assim, ela funciona como um “timer”.

Por exemplo: uma aplicação poderia utilizar os serviços de “Despertador” para receber uma notificação a cada cinco minutos. A aplicação então poderia checar se existem arquivos abertos e se eles estão em uso. Se, nos últimos cinco minutos, os arquivos permaneceram abertos mas não houve qualquer atividade por parte do operador do sistema, a aplicação pode simplesmente fechar os arquivos ou tomar qualquer outra atitude.

Outro exemplo: em um sistema operacional que não dispusesse de um mecanismo eficiente de sinais ou mensagens, uma aplicação poderia usar os serviços de “Despertador” para, a cada cinco segundos, ler o arquivo de configurações “*aplicacao.conf*” (ou

“*aplicacao.ini*”). Com isto, a aplicação ficaria sabendo que seus parâmetros de funcionamento foram alterados e, a partir daí, poderia reorientar o modo de execução de alguns dos seus procedimentos.

Enfim: a classe “Despertador” realiza o trabalho genérico de controlar intervalos de tempo, garantindo assim que periodicamente seja disparada uma chamada de função. Essa chamada de função cumpre o papel de uma **mensagem** que notifica a aplicação sobre a ocorrência de um **evento** (e, no caso da classe “Despertador”, o evento é: “*novamente foi atingido o fim do período estipulado*”).

“Despertador” conhece tudo sobre controle de tempo e conhece também as **características de tipo** da função que deve ser disparada (isto é: quais são os seus parâmetros e qual é o seu retorno). Mas “Despertador” não conhece o trabalho específico que deve ser feito pela função, pois, como vimos nos dois exemplos acima, isso depende da aplicação específica –podendo implicar em atividades completamente diferentes.

Por isso a classe base deve declarar uma função **virtual** para que esta seja **redefinida** em derivadas.

Implementação do exemplo:

```
#include <iostream>
#include <cstdlib>
#include <ctime>

class Despertador
{
public:
    Despertador()
        // vtable[0] = &Despertador::Despertar // na base
    {
    }

    virtual bool Despertar( int nVez ) // função virtual
    {
        return false;
    }

public:
    void Iniciar( int Intervalo )
    {
        int Vez = 0;
        do
        {
            Vez++;
            Pausa(Intervalo);

        } while ( Despertar( nVez ) ); // chamada à função virtual:
                                         // O EVENTO É DISPARADO
    }

    void Pausa ( int Intervalo )
    {
        clock_t Fim =((clock_t)Intervalo * CLOCKS_PER_SEC)
```

```
        + clock());
    while( Fim > clock() );
}
};

// DERIVADA: irá redefinir a função virtual "Despertar"
class AcordaParaImprimir : public Despertador
{
    AcordaParaImprimir()
        : // vtable[0] = &AcordaParaImprimir::Despertar
    {}

    bool Despertar(int nVez) // redefinindo esta função,
    {                          // irá RESPONDER AO EVENTO
        cout << "Acordei. Vou Imprimir" << endl;
        return (nVez<10);
    }
};

int main()
{
    AcordaParaImprimir DaUmTempo; // objeto é criado a partir da derivada
    DaUmTempo.Iniciar( 1 );
    return 0;
}
```

### 11.3.4.1 • Funções virtuais puras e classes abstratas.

Observe que no exemplo da classe “*Despertador*”, a função virtual “*Despertar*” da classe-base simplesmente não faz nada, tendo apenas uma linha de retorno (“*return false*”), que interromperá a pausa na primeira vez em que ela for usada.

Se ela fizesse alguma coisa minimamente útil, então a função de mesmo nome da derivada poderia até, se fosse conveniente, reaproveitar o seu código fazendo uma chamada direta à função-base.

Mas não é esse o caso.

Assim poderíamos declarar essa função como uma **virtual-pura**.

Uma função virtual pura simplesmente não existe na classe base. É apenas uma interface (um protótipo). Contudo o fato de que ela tenha sido declarada faz com que a tabela de ponteiros para funções virtuais (*vtable*) ganhe mais um elemento.

Só que as entradas na matriz de ponteiros para as funções virtuais puras deverão ser assinalada como nulas (já que esse tipo de função não tem corpo e assim sendo não existirá qualquer endereço válido).

Por isso se uma classe tem pelo menos uma função virtual pura ela passa a ser considerada como uma **classe abstrata**.

Isto é: não podemos declarar objetos a partir dessa classe e sim, **apenas**, a partir de classes **derivadas** que tenham **redefinido a função virtual pura criando um corpo** (código endereçável) para ela

Logo, na derivada, essa função não poderá ser também virtual pura (do contrário continuaríamos com uma chamada a um ponteiro de função cujo conteúdo é um endereço inválido).

E, se isso acontecer, então teremos que criar uma derivada dessa primeira derivada, até que a função adquira um corpo endereçável.

Enfim: só uma classe que **implemente** a função permitirá que sejam criados objetos a partir dela.

Desse modo a função “*Despertar*” da classe “*Despertador*” poderia ter sido declarada assim:

```
virtual bool Despertar ( ) = 0 ;
```

A especificação “**igual a zero**” no final do protótipo da função indica que ela é uma virtual pura; não terá corpo nesta classe e o seu endereço é assim inválido (zero, isto é, nulo).

### 11.3.4.2 • Exercício: a classe *RelatPadrao*

Veja o código fonte deste exercício no diretório do instrutor:



A solução completa está em: “**Exercicios/virtuais/Relatorio**”



#### 11.3.4.2.a • Solução: declaração, implementação e uso da classe RelatPadrao.

A classe RelatPadrao fornece a infraestrutura básica para a impressão de relatórios. Só pode ser usada a partir de uma derivada, já que contém uma função virtual-pura, a função "ImprDetalhe", a qual deverá ser redefinida para imprimir as linhas de detalhe.

**Por exemplo:**

```
class Relato : public RelatPadrao
{
    public:

    // redefine a virtual-pura para impressão de detalhes:

    bool ImprDetalhe();
};
```

Além disso, antes de começar a impressão podemos definir linhas de cabeçalho e/ou de rodapé, conforme o exemplo abaixo:

```
int main()
{
    RelatPadrao::StringList Cabec; // StringList: sinônimo para list<string>
    Cabec.insert(Cabec.end(), "Título Geral do Relatório - Folha =#000#");
    // podemos acrescentar mais linhas com insert;
    // OBS: por convenção da classe, um # finalizado com outro #
    // marca a posição em que deve ser inserido o número da FOLHA.

    Relato Rel;

    // cabeçalho com um fio horizontal separador
    // e uma linha em branco para separação:
    Rel.Cabecalho(Cabec, true, 1);

    // A função abaixo ("Imprime") ficará em loop,
    // imprimindo cabeçalho e rodapé no momento adequado, e chamando
    // a virtual-pura "ImprDetalhe" até que esta retorne falso,
    // quando então o loop será finalizado e "Imprime" retornará:
    Rel.Imprime("LPT1");
    return 0;
}
```

Além disso, a classe derivada poderia também redefinir as virtuais(não-puras) "ImprCabecalho" e "ImprRodape", caso precisasse modificar o comportamento padrão dessas funções-base as quais já imprimem o cabeçalho e o rodapé a partir das "StringList's" informadas nas funções "Cabecalho" e "Rodape".

A solução está dividida em 3 arquivos:

- a. declaração da classe "RelatPadrao" (**Relat.h**)
- b. implementação da classe (**Relat.cpp**)
- c. um exemplo de uso (**Testa.cpp**).

### 11.3.5 ▪ Precauções ao usar funções virtuais.

#### 11.3.5.1.a ▪ 1 - Nunca preencher um objeto com métodos de baixo nível.

- Ao usar uma única função virtual, uma classe já passa a ter um **membro de dados oculto** (a **vtable**). Por isso **nunca** utilize uma prática comum entre programadores **C** :

```
Data Qualquer;
memset ( &Qualquer , 0x0 , sizeof(Data) );
```

- Isto já é ruim pois anula o encapsulamento.
- Mas mesmo o código abaixo, também **não** deve ser escrito:

```
Data::Data ( ) // construtora.
{
    memset ( this , 0x0 , sizeof( Data ) );
}
```

- Nos **dois casos** a **vtable** foi **zerada logo após ter sido preenchida**, levando o programa a executar um endereço inválido na primeira oportunidade em que a virtual for chamada.

#### 11.3.5.1.b ▪ 2 - Nunca esquecer como a vtable é preenchida.

Sempre que usarmos virtuais e, especialmente, **virtuais-puras**, temos que estar conscientes do modo como a vtable é preenchida (lógica de construtoras e destrutora entre classe base e derivada):

- Na **construtora-base** o endereço existente para uma virtual na **vtable** é o endereço da função **membra** da classe **base**.
- Se chamamos uma virtual aí estaremos chamando sempre a função da classe base.
- E, se ela for uma **virtual-pura** o seu **endereço é inválido** (zero, pois a função não existe aqui).



Logo, não se pode chamar uma virtual-pura dentro da construtora-base (erro de compilação), ou, indiretamente, em qualquer outra função da classe que seja chamada pela construtora (erro de execução).

Só **após o retorno da construtora-base**, e antes da execução da construtora-derivada, é que o ponteiro estará preenchido adequadamente com o endereço da função redefinida na derivada.

- Na **destrutora-base** o endereço de uma função **virtual** armazenado na **vtable**, volta a ser o endereço da função **membra** da classe **base**.
- **Então**, o último ponto em que uma virtual **pura** pode ser chamada é a destrutora da **derivada**.
- Pois, após o retorno da destrutora derivada ela será completamente destruída. Assim, após este retorno, e antes de ingressar na destrutora-base, o endereço de uma virtual na **vtable** volta a ser o endereço da classe base. E, sendo **pura**, o **endereço é inválido**.



Então, não podemos chamar uma virtual-pura na destrutora-base ou em função por ela chamada.

## 11.4 • Questões para revisão

Responda às questões abaixo. **Em seguida, compare suas respostas** com as respostas localizadas no **Anexo B-7, página 471**. **Caso não entenda**, encaminhe as dúvidas ao instrutor.

- responda à pergunta que está no comentário do código abaixo:

```
class X
{
    .....
};
class Y : public X ❸ // O que significa esta declaração?
{
    .....
};
```

---



---



---



---

- responda à pergunta que está no comentário do código abaixo:

```
class X
{
    virtual bool Funcao( ) ❹ // Que função é esta e para que serve?
    {
        .....
    }
};
```

---



---



---

- responda à pergunta que está no comentário do código abaixo:

```
class X
{
    X( ) { }
};
class Y : public X
{
    Y( ) { }
};


int main ( )
{
    Y meu_Y; ❺ // Que funções serão executadas aqui ?
               // E em que ORDEM ?
    .....
}
```

- responda à pergunta que está no comentário do código abaixo:

```

class X
{
    int Imprime ( ) {
        .....
    }
};

class Y : public X
{
    int Imprime( )
    {
        .....
        X::Imprime();
    }
};

int main ( )
{
    Y meu_Y;
    meu_Y.Imprime( ) ;  // Que funções (e de quais classes)
                                // serão chamadas aqui ?
                                // E em que ORDEM ?
    .....
}

```

- responda à pergunta que está no comentário do código abaixo:


```

class X
{
    virtual bool Funcao( )
    {
        .....
    }

    int Imprime ( )
    {
        Funcao( ) ;
        .....
    }
};

class Y : public X
{
    bool Funcao( ) {
        .....
    }
};

```

```
int main ( )
{
    Y meu_Y ;
    meu_Y.Imprime( ) ;  // Que funções (e de quais classes)
                        // serão chamadas aqui ?
                        // E em que ORDEM ?
    .....
}
```

---


---

---

---

---

- responda à pergunta que está no comentário do código abaixo:

```
template <typename T> class X
{
    T Valor ;  // Qual é o tipo do membro de dados "Valor" ?
    .....
};
```

---

---

## • Capítulo 12

### ▪ Tratamento de Exceções

---

12.1 • Usando throw ... try {...} catch(...) {...}.....	319
12.2 • Usando um tratador <i>default</i> .....	320
12.3 • <code>std::exception</code> .....	321
12.4 • Derivadas padrão de <code>std::exception</code> .....	324

## 12.1 • Usando throw ... try {...} catch(...) {...}

Um desvio de fluxo especial é usado para o tratamento de exceções.

Trata-se da combinação **throw ... try {...} catch {...}**

Se uma determinada função é usada de modo indevido, recebendo uma ordem impossível de cumprir (por exemplo, uma tentativa de acessar uma memória não alocada), ela pode exigir que esse erro seja tratado como uma exceção por quem tenha chamado a função.

Assim a função utilizará **throw**, que dispara uma chamada a um tratador de exceções.

O trecho do código responsável pela chamada incorreta, deveria ter se precavido, apresentando um tratador de exceções para que ele possa ser chamado em caso de violação.

Assim:

```
try
{
    // chama a função crítica a qual chamará throw em caso de violação
}

catch(...) //em caso de violação throw desviará o processamento para cá:
{
    // código para tratamento da exceção.
}
```

O **try** posiciona o bloco associado ao **catch** como sendo o código responsável pelo tratamento de exceções que sejam provocadas por qualquer instrução contida no bloco associado ao próprio **try**.

Assim, se, em uma das instruções dentro do bloco associado ao **try**, for chamada uma determinada função e esta função constatar uma violação e fizer uma chamada a **throw**, imediatamente será executado o bloco de código associado ao **catch**.

### *Exemplo:*

```
#include <iostream>
#include <string>

using namespace std;


// classe com informações para tratamento de exceções
class ErrDim
{
public:
    string m_Descr;    // descrição da exceção
    string m_Origem;  // onde ela ocorreu
};

// classe onde será disparada a exceção:
class Generica
```

```

{
    public:
        enum {MAX_DIM=10};
    private:
        int Matriz[MAX_DIM];
    public:
        void Atribuir(int indice, int valor);
};

void Generica::Atribuir(int indice, int valor)
{
    if ( indice <= MAX_DIM )
        Matriz[indice] = valor;
    else // Impossível continuar!!!
    {
        ErrDim edim;
        edim.m_Descr = "indice invalido";
        edim.m_Origem = "Generica::Atribuir/excecoes.cpp";

        throw edim; //  chama um tratador de exceção
                    // passando "edim" como parâmetro
    }
}

int main()
{
    Generica matr;

    try
    {
        matr.Atribuir(20, 50); // uso incorreto! estouro de dimensão.
    }

    catch( ErrDim & ed ) // exceções informadas por um objeto ErrDim
    {
        cout << "Descricao: " << ed.m_Descr << endl;
        cout << "Origem: " << ed.m_Origem << endl;
    }

    catch(...) // qualquer outra exceção
    {
        cout << "Erro não identificado" << endl;
    }

    return 0;
}

```

## 12.2 • Usando um tratador *default*

Caso a função **main** não tivesse tratado a exceção, o programa seria interrompido. Contudo, a classe **Generica** poderia ter previsto essa situação, definindo um tratador *default* que seria acionado se o disparo efetuado por *throw* não encontrasse nenhum tratador no ponto onde ocorreu a violação de acesso.

Isto pode ser feito cadastrando-se um ponteiro para uma função tratadora, através da função **set\_terminate**:



```

void term_function()
{
    cout << "execacao nao foi tratada" << endl;
    exit(-1);
}
void Generica::Atribuir(int indice, int valor)
{
    if ( indice <= MAX_DIM )
        Matriz[indice] = valor;

    else // Impossível continuar!!!
    {
        ErrDim edim;
        edim.m_Descr ="indice invalido";
        edim.m_Origem="Generica::Atribuir/excecoes.cpp";

        // define abaixo um tratador default, que será usado
        // se nenhum tratador responder ao throw

        set_terminate(term_function); // ➡ term_function será o // tratador
        default.

        // a função term_function
        // será chamada se não houver resposta para o disparo abaixo:

        throw edim; // chama um tratador de exceção
    }
}

```

## 12.3 • std::exception

O exemplo acima funciona, mas o **ideal é utilizar uma *interface* bem conhecida para o tipo do argumento que será usado quando uma exceção é lançada.**

Para isso temos **std::exception**.

Para personalizá-la podemos criar uma derivada, ou, **melhor, usar uma derivada já existente na própria biblioteca padrão.**

Neste exemplo, usaremos uma derivada própria. No próximo, usaremos uma derivada já existente na biblioteca. Basicamente, será preciso redefinir a virtual "**what**" da base.

```

#include <iostream>
#include <string>
#include <exception>
#include <cstdlib>

// emite uma mensagem final
void MsgFinal()
{
    std::cout << "\n<enter> p/sair\n";
    std::cin.get();
}

// tipo parametrizado    valor constante
//                      \\\          \\\
template <typename T, size_t Dim> class VetorFixo
{
private:
    T m_Vetor[ Dim ]; // vetor de "T" com dimensões fixas

```

```

// Classe aninhada para tratamento de exceções
// (aninhada, pois só serve para trabalhar com "VetorFixo").
// As funções "get" e "set" de VetorFixo<> usarão esta classe
// para disparar exceções:
class Exception : public std::exception // deriva da classe padrão
{
    da STL

    friend class VetorFixo<T, Dim> ;
    // ações que podem provocar uma exceção
    // (funções get/set de "VetorFixo"):
    enum Action {actGet, actSet } ;
    Action m_action ;

    // construtora private
    // (poderá ser usada apenas na sua "friend" VetorFixo<>):
    Exception( Action action ) : m_action ( action ) {}

public:
    // redefine virtual DA BASE:
    virtual const char* what() const throw()
        // o "throw()" aqui - sem especificação de tipo -
        // adverte que esta função não dispara exceptions
    {
        switch ( m_action )
        {
            case actGet : return "VetorFixo::GET : indice incorreto" ;
            case actSet : return "VetorFixo::SET : indice incorreto" ;
            default:
                return std::exception::what();
                // algo inesperado aqui ...
        }
    }
} ;

// funções estáticas que redefinem os tratadores
// "default" para exceptions:
// se ocorrer um "throw" e ele não for capturado
// em um "try/catch",
// os tratadores serão estes:
static void excecaoDefault_getFunc ()
{
    std::cout << "\nTERMINATE: VetorFixo::GET: indice incorreto :"
                << " (sem try/catch)\n" ;
    MsgFinal();
    exit(-1); // encerra aplicação
}
static void excecaoDefault_setFunc ()
{
    std::cout << "\nTERMINATE: VetorFixo::SET: indice incorreto :"
                << " (sem try/catch)\n" ;
    MsgFinal();
    exit(-1); // encerra aplicação
}


public:
    VetorFixo(){}


    inline size_t Size () { return Dim ; }

    // verificam índice:
    inline const T & Get( size_t nIndex )

```

```

{
    {
        if ( nIndex >= Dim )
        {
            // redefine o tratador default
            // (caso a exceção não seja capturada em um try/catch)
            std::set_terminate( excecaoDefault_getFunc );
             // objeto de erro personalizado
            Exception err ( Exception::actGet );
            throw err;
        }
        return m_Vetor [ nIndex ];
    }
}

inline void Set( size_t nIndex, const T & newValue )
{
    if ( nIndex >= Dim )
    {
        // redefine o tratador default
        // (caso a exceção não seja capturada em um try/catch)
        std::set_terminate( excecaoDefault_setFunc );
         // objeto de erro personalizado
        Exception err ( Exception::actSet );
        throw err;
    }
    m_Vetor [ nIndex ] = newValue;
}

// não verificam índice:
inline T & operator[ ] ( size_t nIndex )
{
    return m_Vetor[ nIndex ] ;
}

inline const T & operator[ ] ( size_t nIndex ) const
{
    return m_Vetor[ nIndex ] ;
}

};

void TrataExcecaoDefault ()
{
    std::cout << "impossivel continuar" << std::endl;
    exit(-1); // encerra aplicação
}

int main()
{
    // esse "set_terminate" aqui é genérico demais...
    // nunca saberemos onde ocorreu o problema.
    std::set_terminate( TrataExcecaoDefault );

    const int viDim = 5 ;
    VetorFixo< int , viDim > vi;

    vi.Set ( 2, 1); // OK: acessa elemento alocado

    try // provoca exception com a função "Get":
    {
        std::cout << vi.Get( viDim ) << "\n"; // acessa elemento
                                                // não alocado...
    }

```

```

    }
    catch( std::exception & err )
    {
        std::cout << "Erro: " << err.what() << "\n";
    }
    catch(...) // se o tipo do erro não for "std::exception"
                                                    // então vai cair aqui
    {
        std::cout << "algo desconhecido, talvez terrível, ocorreu\n";
    }

    try // provoca exception com a função "Set":
    {
        vi.Set( viDim, 10 ) ; // acessa elemento não alocado...
    }
    catch( std::exception & err )
    {
        std::cout << "Erro: " << err.what() << "\n";
    }

    // Acessa elemento não alocado sem "try/catch".
    // O "terminate" será usado:
    int x = vi.Get( viDim ) ;
    std::cout << x << "\n";

    MsgFinal();
    return 0;
}

```

## 12.4 • Derivadas padrão de std::exception

Para o exemplo acima, teria sido mais simples usar uma derivada já existente na biblioteca. As derivadas já existentes permitem cobrir erros comuns (runtime, range, etc) e podemos personalizar a mensagem que é devolvida pela função virtual **what**.

Caso houvesse mais coisas a personalizar, poderíamos criar uma derivada própria, como no exemplo acima.

Mas para esse exemplo temos uma derivada padrão que se aplica perfeitamente: **range\_error**.

```

#include <iostream>
#include <string>
#include <stdexcept>

#include <cstdlib>

// emite uma mensagem final
void MsgFinal()
{
    std::cout << "\n<enter> p/sair\n";
    std::cin.get();
}

// tipo parametrizado    valor constante
//          \\\          \\\
template <typename T, size_t Dim> class VetorFixo
{
private:

```



```

T m_Vetor[ Dim ]; // vetor de "T" com dimensões fixas

// funções estáticas que redefinem os tratadores
// "default" para exceptions:
// se ocorrer um "throw" e ele não for capturado em um "try/catch",
// os tratadores serão estes:
static void excecaoDefault_getFunc ()
{
    std::cout << "\nTERMINATE: VetorFixo::GET: indice incorreto : "
                << " (sem try/catch)\n" ;
    MsgFinal();
    exit(-1); // encerra aplicação
}
static void excecaoDefault_setFunc ()
{
    std::cout << "\nTERMINATE: VetorFixo::SET: indice incorreto : "
                << " (sem try/catch)\n" ;
    MsgFinal();
    exit(-1); // encerra aplicação
}

public:
    VetorFixo(){}

    inline size_t Size () { return Dim ; }

    // verificam índice:
    inline const T & Get( size_t nIndex )
    {
        {
            if ( nIndex >= Dim )
            {
                // redefina o tratador default
                // (caso a exceção não seja capturada em um try/catch)
                std::set_terminate( excecaoDefault_getFunc );
                 // objeto de erro, com mensagem para "what":
                std::range_error err("VetorFixo::GET - indice incorreto");
                throw err;
            }
            return m_Vetor [ nIndex ];
        }
    }
    inline void Set( size_t nIndex, const T & newValue )
    {
        {
            if ( nIndex >= Dim )
            {
                // redefina o tratador default
                // (caso a exceção não seja capturada em um try/catch)
                std::set_terminate( excecaoDefault_setFunc );
                 // objeto de erro, com mensagem para "what":
                std::range_error err ("VetorFixo::SET - indice incorreto");
                throw err;
            }
            m_Vetor [ nIndex ] = newValue;
        }
    }

    // não verificam índice:
    inline T & operator[ ] ( size_t nIndex )

```

```

    {
        return m_Vetor[ nIndex ] ;
    }
    inline const T & operator[ ] ( size_t nIndex ) const
    {
        return m_Vetor[ nIndex ] ;
    }
};

int main()
{
    const int viDim = 5 ;
    VetorFixo< int , viDim > vi;

    vi.Set ( 2, 1); // OK: acessa elemento alocado

    try // provoca exception com a função "Get":
    {
        std::cout << vi.Get( viDim ) << "\n"; // acessa elemento
                                                // não alocado...
    }
    catch( std::exception & err )
    {
        std::cout << "Erro: " << err.what() << "\n";
    }
    catch(...) // se o tipo do erro não for "std::exception"
                                                // então vai cair aqui
    {
        std::cout << "algo desconhecido, talvez terrível, ocorreu\n";
    }

    try // provoca exception com a função "Set":
    {
        vi.Set( viDim, 10 ) ; // acessa elemento não alocado...
    }
    catch( std::exception & err )
    {
        std::cout << "Erro: " << err.what() << "\n";
    }

    // Acessa elemento não alocado sem "try/catch".
    // O "terminate" será usado:
    int x = vi.Get( viDim ) ;
    std::cout << x << "\n";

    MsgFinal();
    return 0;
}

```

---

## • Capítulo 13

### ▪ A biblioteca de tempo Chrono

---

A biblioteca Chrono é utilizada para o controle do tempo. Todos os recursos desta biblioteca estão disponíveis através do namespace `std::chrono`.

Chrono trabalha com 3 (três) conceitos principais:

- **Durações:** medem um tempo que passou, por exemplo: 10 segundos, 2 horas, etc. As durações são representadas por um objeto `duration<>` que se baseia em um contador e uma precisão do período, por exemplo, 10 é o contador e a precisão é segundos.
- **Pontos no tempo (time points):** é a representação de um momento específico no tempo, por exemplo a data de nascimento, a hora do próximo trem, etc. São representados pelo objeto `time_point<>` e usa uma duração relativa a uma época (que é um ponto fixo no tempo comum a todos os `time_points` que usam o mesmo clock).
- **Clocks:** Um framework que relaciona um `time_point` a um tempo físico real. A biblioteca providencia pelo menos 3 (três) clocks que disponibilizam meios para expressar o tempo atual como um `time_point`: `system_clock`, `steady_clock` e `high_resolution_clock`.

### Exemplo:

Neste exemplo, armazenamos a data atual em um `time_point`. Fazemos um processamento qualquer e em seguida pegamos a data atual novamente. Depois disso subtraímos um do outro para obter o tempo decorrido entre o início e o fim do processamento:

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

int main()
{
    //Pega hora final
    steady_clock::time_point inicio = steady_clock::now();

    for(int n = 0; n < 30000; ++n)
        cout << '.';

    //Pega hora atual
    steady_clock::time_point fim = steady_clock::now();
```



```
double tempo = duration_cast<milliseconds>(fim - inicio).count();  
cout << "\nmilliseconds: " << tempo << '\n';  
return 0;  
}
```

## • Capítulo 14

### ▪ Threads

---

14.1 • A classe <code>std::thread</code> (C++11).....	331
14.2 • Async Thread (C++11).....	332
14.3 • Mutex (C++11).....	334
14.4 • Recursive mutex (C++11).....	336
14.5 • Timed mutex (C++11).....	336
14.6 • Recursive timed mutex (C++11).....	337
14.7 • Lock guard (C++11).....	338
14.8 • Unique lock (C++11).....	339
14.9 • Future/Promise (C++11).....	339
14.10 • Call once (C++11).....	342
14.11 • TLS – Thread Local Storage (C++11).....	343
14.12 • Atomic (C++11).....	343

## 14.1 • A classe std::thread (C++11)

Um objeto std::thread executa um objeto que pode ser chamado (função, lambda,...) passado como primeiro argumento de forma assíncrona.

### Exemplo 1:

```
#include <iostream>
#include <thread>

using namespace std;

void funcao()
{
    cout << "Thread disparado.\n\n";
}

int main()
{
    thread th(funcao);
    th.join();
    return 0;
}
```

### Exemplo 2:

```
#include <iostream>
#include <thread>
#include <chrono>
#include <fstream>

using namespace std;
using namespace chrono;

void pause_thread(int n)
{
    this_thread::sleep_for(milliseconds(n));
    cout << "Pausa de " << (n / 1000) << " segundos finalizada.\n";
}

int main()
{
    cout << "Disparando 3 threads de forma independente...\n";
    thread (pause_thread, 1000).detach();
    thread (pause_thread, 2000).detach();
    thread (pause_thread, 3000).detach();

    cout << "O main thread vai parar por 5 segundos.\n\n";
    pause_thread(5000);

    return 0;
}
```

## 14.2 • Async Thread (C++11)

Uma outra forma de executar uma função de forma assíncrona, é por meio da função `std::async()`. A vantagem é que um thread iniciado desta forma pode retornar um valor ou uma exceção para o thread criador.

### Exemplo 1:

```
#include <iostream>
#include <thread>
#include <future>

using namespace std;

double soma(double a, double b)
{
    return a+b;
}

int main()
{
    auto result = async(soma, 10.0, 20.0);
    cout << result.get() << '\n';
    return 0;
}
```

Neste exemplo, uma operação assíncrona foi iniciada através da chamada à função `std::async`. Quando precisarmos do valor de retorno da função, usamos o método `get()` do objeto retornado por `async()`. Mas, o que a função `async` retorna? A resposta é um objeto do tipo `std::future<>` que foi criado para esta finalidade. A chamada à função `std::async` poderia ter sido feita assim:

```
#include <iostream>
#include <thread>
#include <future>

using namespace std;

double soma(double a, double b)
{
    return a+b;
}

int main()
{
    future<double> result = async(soma, 10.0, 20.0);
    cout << result.get() << '\n';
    return 0;
}
```

O template `std::future<>` tem como argumento o nome do tipo que o retorno da função iniciada por `std::async` retornar. Neste caso, a função

soma retorna double, e por esse motivo o tipo passado como argumento para `std::future<>` deve ser double.

## Exemplo 2:

```
#include <iostream>
#include <thread>
#include <future>
#include <vector>

using namespace std;

double calculo()
{
    int t = 50;
    double result = 0;
    for(int n = 0; n < t; ++n)
        result += n;

    cout << "Thread id: " << this_thread::get_id() << " = " << result;
    return result;
}

int main()
{
    vector<future<double>> vfuture;
    vfuture.resize(10);

    for(int n = 0; n < 10; ++n)
        vfuture[n] = async(calculo);

    for(int n = 0; n < 10; ++n)
        cout << " | Future:[" << n << "] = " << vfuture[n].get() << '\n';

    return 0;
}
```

## Exemplo 3:

```
#include <iostream>
#include <thread>
#include <mutex>
#include <future>
#include <vector>

using namespace std;

int calculoThread = 0;

mutex mu;

int calcular()
{
    lock_guard<mutex> lock(mu);
    //Para diferenciar o valor de cálculo de cada thread
    ++calculoThread;
    int result = 0;
```

```

for(int i = 0; i < 10; ++i)
    result += i;

cout << "Resultado gerado pelo thread: " << this_thread::get_id()
    << ", resultado: " << (result + calculoThread) << '\n';

return result + calculoThread;
}

int main()
{
    vector<future<int>> vecFuture;//Cria vetor de future
    vecFuture.resize(10);

    //Os threads serão disparados, calculados e future vai
    //armazenar seu valor de retorno.
    for(int i = 0; i < 10; ++i)
        vecFuture[i] = async(launch::async, calcular);

    //Para dar tempo de todos os threads serem processados
    this_thread::sleep_for(chrono::seconds(2));

    //Capturando o valor processado pelos threads atraves de future
    for(int i = 0; i < 10; ++i)
        cout << "Pegando resultado[" << i << "]" << ", " << vecFuture[i].get() << '\n';

    return 0;
}

```

Será que a função `std::async` executou mesmo a função `soma()` do exemplo anterior de forma assíncrona? Pode ser. Isso porque neste caso, a implementação decide se deve iniciar a execução imediatamente, ou se é melhor esperar até que o método `get()` seja chamado para executar a função `soma()` de forma síncrona.

Podemos escolher como executar o callable object passando um argumento a mais para a função `std::async()`:

- `auto result = std::async(std::launch::async,...);` // assíncrono
- `auto result = std::async(std::launch::deferred,...);` // síncrono

## 14.3 • Mutex (C++11)

A execução concorrente entre threads pode ser um problema quando eles precisam acessar dados compartilhados. Em um determinado momento os threads poderão acessar simultaneamente esses dados, e obter resultados corrompidos.

Para resolver esse tipo de problema, podemos usar mutex (exclusão mutual), que bloqueia o acesso a um recurso enquanto necessário e desbloqueia quando terminar. As classes disponíveis para mutex são:

- `std::mutex` `std::timed_mutex`
- `std::recursive_mutex`
- `std::recursive_timed_mutex`

Geralmente, um mutex é gerenciado por uma das classes RAII (Resource Acquisition is Initialization) da biblioteca padrão, são elas:

- `std::lock_guard<>`: Bloqueia o mutex na construtora e desbloqueia na destrutora. Nenhuma outra operação é permitida.
- `std::unique_lock<>`: Mais flexível que o `std::lock_guard<>`, pode adquirir o travamento sobre o mutex depois de construído e desbloquear antes da destrutora. Ele também pode ser movido.

Não pode fazer dois locks simultâneos.

```
#include <iostream>
#include <thread>
#include <mutex>

using namespace std;

mutex mu;

void escreva()
{
    for(int i = 0; i <= 10; ++i)
    {
        //cout esta sendo compartilhado por 2 threads e
        //por isso devemos colocar o mutex aqui.
        mu.lock();
        cout << "Thread: " << this_thread::get_id()
              << " imprimindo: " << i << '\n';
        //E destravar aqui.
        mu.unlock();
    }
}

int main()
{
    thread th1(&escreva), th2(&escreva);

    th1.join();
```

```

th2.join();

return 0;
}

```

## 14.4 • Recursive mutex (C++11)

Podemos fazer dois locks simultâneos.

```

#include <iostream>
#include <thread>
#include <mutex>

using namespace std;

recursive_mutex mu;

void escreva()
{
    for(int i = 0; i <= 10; ++i)
    {
        //cout esta sendo compartilhado por 2 threads e
        //por isso devemos colocar o recursive_mutex aqui.
        mu.lock();
        mu.lock();

        cout << "Thread: " << this_thread::get_id()
             << " imprimindo: " << i << '\n';

        //E destravar aqui.
        mu.unlock();
        mu.unlock();
    }
}

int main()
{
    thread th1(&escreva), th2(&escreva);

    th1.join();
    th2.join();

    return 0;
}

```

## 14.5 • Timed mutex (C++11)

Tenta travar o mutex por um determinado tempo.

```

#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>

```



```
using namespace std;
using namespace chrono;

timed_mutex mu;

void escreva()
{
    for(int i = 0; i <= 10; ++i)
    {
        //Tenta travar por 2 segundos
        mu.try_lock_until(steady_clock::now() + seconds(2));

        cout << "Thread: " << this_thread::get_id()
              << " imprimindo: " << i << "\n";

        //E destravar aqui.
        mu.unlock();
    }
}

int main()
{
    thread th1(&escreva), th2(&escreva);

    th1.join();
    th2.join();

    return 0;
}
```

## 14.6 • Recursive timed mutex (C++11)

É a união de `timed_mutex` com `recursive_mutex`.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>

using namespace std;
using namespace chrono;

recursive_timed_mutex mu;

void escreva()
{
    for(int i = 0; i <= 10; ++i)
    {
        //Tenta travar por 2 segundos
        mu.try_lock_until(steady_clock::now() + seconds(2));
        mu.try_lock_until(steady_clock::now() + seconds(2));

        cout << "Thread: " << this_thread::get_id()
              << " imprimindo: " << i << "\n";

        //E destravar aqui.
        mu.unlock();
    }
}
```

```

        mu.unlock();
    }
}

int main()
{
    thread th1(&escreva), th2(&escreva);

    th1.join();
    th2.join();

    return 0;
}

```

## 14.7 • Lock guard (C++11)

Quando criamos um mutex corremos o risco de que haja um if, um return, ou até mesmo uma exceção no meio do caminho antes de chegar no unlock e isso causaria dead lock no programa. Para evitar isso, foi criado o `lock_guard`, que trava o mutex na construtora e destrava na destrutora, garantindo assim que o mutex sempre será travado e depois destravado.

```

#include <iostream>
#include <thread>
#include <mutex>

using namespace std;

mutex mu;

void escreva()
{
    for(int i = 0; i <= 10; ++i)
    {
        //lock_guard travará mu na construtora
        lock_guard<mutex> lockGuard(mu);

        cout << "Thread: " << this_thread::get_id()
              << " imprimindo: " << i << '\n';
    }
}

int main()
{
    thread th1(&escreva), th2(&escreva);

    th1.join();
    th2.join();

    return 0;
}

```

## 14.8 • Unique lock (C++11)

Nos da liberdade para travar/destravar quando quisermos, mas é garantido que sempre que passar pela destrutora, será destravado.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>

using namespace std;
using namespace chrono;

mutex mu;
int global = 0;

void escreva()
{
    for(int i = 0; i <= 10; ++i)
    {
        //lock_guard trará mu na construtora
        unique_lock<mutex> uniqueLock(mu);
        ++global;
        uniqueLock.unlock();
        //Nos da liberdade para destrava/travar quando quisermos,
        //mas é garantido que sempre que, passar pela
        //destrutora será destravada.
        uniqueLock.lock();
        cout << "Thread: " << this_thread::get_id() << " imprimindo: " << i << '\n';
    }
}

int main()
{
    thread th1(&escreva), th2(&escreva);

    th1.join();
    th2.join();

    this_thread::sleep_for(seconds(2));
    cout << "\n\n\nGlobal: " << global << '\n';

    return 0;
}
```

## 14.9 • Future/Promise (C++11)

Vimos anteriormente, que um objeto `std::future<>` é utilizado para pegar o valor retornado por uma função executada por `std::async<>`, mas esta não é a única maneira de se disponibilizar dados para um objeto `std::future<>`. Podemos também usar um objeto `std::promise<>` para esta finalidade. Cada objeto `std::promise<>` esta ligado exclusivamente a um objeto `std::future<>`. O thread que tiver acesso ao objeto `std::future<>`

pode esperar até que os dados estejam prontos, enquanto o thread que tem acesso ao objeto `std::promise` correspondente ao `std::future<>` pode chamar o método `set_value()` para disponibilizar os dados.

## 1. Exemplo:

```
#include <iostream>
#include <thread>
#include <future>

using namespace std;

template<typename T>
void imprimirFuture(future<T> &value)
{
    cout << value.get() << '\n';
}

int main()
{
    promise<int> p;
    future<int> ft = p.get_future();
    thread th(imprimirFuture<int>, ref(ft));

    p.set_value(5);
    th.join();

    return 0;
}
```

## 2. Exemplo:

```
#include <iostream>
#include <thread>
#include <future>

using namespace std;

template<typename T>
void imprimirFuture(future<T> &value)
{
    try
    {
        cout << value.get() << '\n';
    }
    catch(int e)
    {
        cout << "codigo do erro: " << e << endl;
    }
}

int main()
{
    promise<int> p;
    future<int> ft = p.get_future();
    thread th(imprimirFuture<int>, ref(ft) );
}
```

```
try
{
    throw 5;
}
catch(...)
{
    p.set_exception( current_exception() );
}

th.join();
return 0;
}
```

Neste exemplo, o thread que contém o objeto `std::promise<>` e que deveria gerar dados para o `std::future<>`, gerou uma exceção que foi capturada e passada para o `std::future<>` através do método `set_exception()`.

### 3. Exemplo:

```
##include <iostream>
#include <thread>
#include <future>

using namespace std;

int tarefa01()
{
    return 1;
}

void tarefa02(promise<int> &p)
{
    p.set_value(2);
}

int main()
{
    future<int> f1 = async(launch::async, tarefa01);
    promise<int> p;
    future<int> f2 = p.get_future();
    thread(tarefa02, ref(p)).detach();

    cout << "f1: " << f1.get() << '\n';
    cout << "f2: " << f2.get() << '\n';

    return 0;
}
```

## Packaged\_task (C++11)

Um objeto `packaged_task` armazena um callable object (função, functor ou lambda) que será executado posteriormente. É muito semelhante a um objeto `std::function`, a diferença é que um objeto `packaged_task` transfere o resultado da chamada ao callable object para um objeto `std::future` que pode ser usado, por exemplo, por um outro thread.

```
#include <iostream>
#include <thread>
#include <future>

using namespace std;

int tarefa01()
{
    return 1;
}

int main()
{
    packaged_task<int> task(tarefa01);
    future<int> f1 = task.get_future();
    thread(move(task)).detach();

    cout << "f1: " << f1.get() << '\n';

    return 0;
}
```

## 14.10 • Call once (C++11)

`call_once` é um template de função que através de uma variável do tipo `once_flag` controla a execução de um callable object, permitindo a execução deste apenas uma vez.

```
#include <iostream>
#include <mutex>

using namespace std;

once_flag flag;

void f()
{
    cout << "funcao f()\n";
}

int main()
{
    call_once(flag, f);
    call_once(flag, f);
    call_once(flag, f);

    return 0;
}
```

## 14.11 • TLS – Thread Local Storage (C++11)

O TLS permite que cada thread tenha sua própria cópia de uma determinada variável. A variável declarada com `thread_local`, será criada quando o thread iniciar e será liberada quando o thread encerrar.

```
#include <iostream>
#include <thread>

using namespace std;

//Gera uma cópia de value com o valor 0 por thread
thread_local int value = 0;

void alterarValue10()
{
    cout << "alterarValue10\n";
    cout << "antes value: " << value << '\n';
    value = 10;
    cout << "depois value: " << value << '\n';
    cout << "thread id: " << this_thread::get_id() << "\n\n";
}

void alterarValue20()
{
    cout << "alterarValue20\n";
    cout << "antes value: " << value << '\n';
    value = 20;
    cout << "depois value: " << value << '\n';
    cout << "thread id: " << this_thread::get_id() << "\n\n";
}

int main()
{
    value = 100;
    thread th1(alterarValue10);
    th1.join();

    alterarValue20();
    cout << "Main\n";
    cout << "final value: " << value << '\n';
    cout << "thread id: " << this_thread::get_id() << '\n';

    return 0;
}
```

## 14.12 • Atomic (C++11)

No padrão C++11, foram introduzidos os tipos atomics, que servem para realizar operações atômicas (indivisíveis, lock-free), de modo a sincronizar a memória entre diferentes threads. Essas mesmas características podem ser obtidas através de mutex, mas uma exclusão mútua pode ser uma operação relativamente cara em termos de latência e do acesso exclusivo. Então, em vez de usar mutex e lock, pode va-

ler a pena usar `atomic`. Entretanto, esteja ciente de que o padrão atômico do padrão **C++11**, não garante que a implementação será lock-free em qualquer plataforma, por isso é melhor saber a sua plataforma e conjunto de ferramentas suportam esse recurso. Você pode chamar `std::atomic< >::is_lock_free` para ter certeza.

## 1. Exemplo:

```
#include <iostream>
#include <atomic>
#include <memory>

using namespace std;

int main()
{
    atomic<int> a;
    atomic<double> b;

    //Verificando se os tipos são lock-free
    cout << boolalpha;
    cout << "int: " << a.is_lock_free() << '\n';
    cout << "double: " << b.is_lock_free() << '\n';

    return 0;
}
```

## 2. Exemplo:

```
#include <iostream>
#include <atomic>
#include <chrono>
#include <thread>

using namespace std;
using namespace chrono;

atomic<int> global(0);

void incrementarGlobal()
{
    for(int i = 0; i < 1000000; ++i)
        ++global;
}

int main()
{
    thread a(incrementarGlobal);
    thread b(incrementarGlobal);

    a.join();
    b.join();

    cout << "global: " << global << '\n';

    return 0;
}
```



## Exercício – Números Primos

1º: Criar novo projeto.

2º: Implementar a seguinte função: `bool ehPrimo(int numero)` que irá dizer se o número é primo ou não.

3º: Na função `main`, utilizando a biblioteca `chrono`, crie um ponto para medir o início do processamento.

4º: Faça chamadas a função `ehPrimo`, com os seguintes números:

- 99971
- 99989
- 99991
- 99992
- 99993
- 99994
- 99995
- 99996
- 99997
- 99998
- 99999
- 100000

5º: Na função `main`, utilizando a biblioteca `chrono`, crie um ponto para medir o fim do processamento.

6º: Exiba a lista dos números acima, dizendo se são ou não são primos.

7º: Exiba o tempo que esses números levaram para serem calculados.

8º: Anote o tempo que o programa levou para calcular os números primos.

**Dica:** Um número é primo quando pode ser dividido por 1 e por ele mesmo, como, por exemplo: 2, 3, 5 e 7.

## Solução – Números Primos

```
#include <iostream>
#include <chrono>

using namespace std;
using namespace chrono;

bool ehPrimo(int numero)
{
    int total = 0;
    for(int i = numero; i > 0; --i)
    {
        if((numero % i) == 0)
            ++total;
        if(total > 2)
            return false;
    }
    return true;
}

int main()
{
    steady_clock::time_point inicio = steady_clock::now();

    bool primo99971 = ehPrimo(99971);
    bool primo99989 = ehPrimo(99989);
    bool primo99991 = ehPrimo(99991);
    bool primo99992 = ehPrimo(99992);
    bool primo99993 = ehPrimo(99993);
    bool primo99994 = ehPrimo(99994);
    bool primo99995 = ehPrimo(99995);
    bool primo99996 = ehPrimo(99996);
    bool primo99997 = ehPrimo(99997);
    bool primo99998 = ehPrimo(99998);
    bool primo99999 = ehPrimo(99999);
    bool primo100000 = ehPrimo(100000);

    steady_clock::time_point fim = steady_clock::now();
    double tempo = duration_cast<milliseconds>(fim - inicio).count();

    cout << boolalpha;
    cout << "99971 primo: " << primo99971 << "\n";
    cout << "99989 primo: " << primo99989 << "\n";
    cout << "99991 primo: " << primo99991 << "\n";
    cout << "99992 primo: " << primo99992 << "\n";
    cout << "99993 primo: " << primo99993 << "\n";
    cout << "99994 primo: " << primo99994 << "\n";
    cout << "99995 primo: " << primo99995 << "\n";
    cout << "99996 primo: " << primo99996 << "\n";
    cout << "99997 primo: " << primo99997 << "\n";
    cout << "99998 primo: " << primo99998 << "\n";
    cout << "99999 primo: " << primo99999 << "\n";
    cout << "100000 primo: " << primo100000 << "\n";

    cout << "\nMilliseconds: " << tempo << "\n\n";
}
```

```
    return 0;  
}
```

## Exercício – Números Primos Threads

1º: Criar novo projeto.

2º: Implementar a seguinte função: `bool ehPrimo(int numero)` que irá dizer se o número é primo ou não.

3º: Na função `main`, utilizando a biblioteca `chrono`, crie um ponto para medir o início do processamento.

4º: Utilizando `asyncs` e `futures`, dispare threads para checar se os seguintes números são primos:

- 99971
- 99989
- 99991
- 99992
- 99993
- 99994
- 99995
- 99996
- 99997
- 99998
- 99999
- 100000

5º: Crie variáveis do tipo `bool` e obtenha os resultados através dos `futures`.

6º: Na função `main`, utilizando a biblioteca `chrono`, crie um ponto para medir o fim do processamento.

7º: Exiba a lista dos números acima, dizendo se são ou não são primos.

8º: Exiba o tempo que esses números levaram para serem calculados.

9º: Anote o tempo que o programa levou para calcular os números primos.

10º: Compare o tempo com o exercício anterior.

**Dica:** Um número é primo quando pode ser dividido por 1 e por ele mesmo, como, por exemplo: 2, 3, 5 e 7.

## Solução – Números Primos Threads

```
#include <iostream>
#include <chrono>
#include <thread>
#include <future>

using namespace std;
using namespace chrono;

bool ehPrimo(int numero)
{
    int total = 0;
    for(int i = numero; i > 0; --i)
    {
        if((numero % i) == 0)
            ++total;
        if(total > 2)
            return false;
    }
    return true;
}

int main()
{
    steady_clock::time_point inicio = steady_clock::now();

    future<bool> future99971 = async(launch::async, ehPrimo, 99971);
    future<bool> future99989 = async(launch::async, ehPrimo, 99989);
    future<bool> future99991 = async(launch::async, ehPrimo, 99991);
    future<bool> future99992 = async(launch::async, ehPrimo, 99992);
    future<bool> future99993 = async(launch::async, ehPrimo, 99993);
    future<bool> future99994 = async(launch::async, ehPrimo, 99994);
    future<bool> future99995 = async(launch::async, ehPrimo, 99995);
    future<bool> future99996 = async(launch::async, ehPrimo, 99996);
    future<bool> future99997 = async(launch::async, ehPrimo, 99997);
    future<bool> future99998 = async(launch::async, ehPrimo, 99998);
    future<bool> future99999 = async(launch::async, ehPrimo, 99999);
    future<bool> future100000 = async(launch::async, ehPrimo, 100000);

    bool primo99971 = future99971.get();
    bool primo99989 = future99989.get();
    bool primo99991 = future99991.get();
```

```
bool primo99992 = future99992.get();
bool primo99993 = future99993.get();
bool primo99994 = future99994.get();
bool primo99995 = future99995.get();
bool primo99996 = future99996.get();
bool primo99997 = future99997.get();
bool primo99998 = future99998.get();
bool primo99999 = future99999.get();
bool primo100000 = future100000.get();

steady_clock::time_point fim = steady_clock::now();
double tempo = duration_cast<milliseconds>(fim - inicio).count();
cout << boolalpha;
cout << "99971 primo: " << primo99971 << '\n';
cout << "99989 primo: " << primo99989 << '\n';
cout << "99991 primo: " << primo99991 << '\n';
cout << "99992 primo: " << primo99992 << '\n';
cout << "99993 primo: " << primo99993 << '\n';
cout << "99994 primo: " << primo99994 << '\n';
cout << "99995 primo: " << primo99995 << '\n';
cout << "99996 primo: " << primo99996 << '\n';
cout << "99997 primo: " << primo99997 << '\n';
cout << "99998 primo: " << primo99998 << '\n';
cout << "99999 primo: " << primo99999 << '\n';
cout << "100000 primo: " << primo100000 << '\n';

cout << "\nMilliseconds: " << tempo << "\n\n\n";

return 0;
}
```

## • Capítulo 15

### ▪ Referência técnica

15.1 • Conversões de tipos.....	352
15.1.1 • Conversões no estilo C.....	352
15.1.2 • Conversões no estilo C++.....	353
15.2 • Tipos agregados e aninhados.....	355
15.2.1 • Tipos aninhados .....	356
15.2.2 • Estruturas aninhadas.....	356
15.2.2.1 • Tipos aninhados: em C eles são apenas indicativos.....	357
15.2.2.2 • Tipos aninhados: C++ não permite o estilo C.....	357
15.2.2.3 • Criação de tipos diretamente no retorno ou nos parâmetros de uma função.....	357
15.3 • Criando tipos através de <i>union</i> .....	358
15.3.1 • Regras para <i>union</i> , específicas de C++.....	359
15.3.2 • Unions com menos restrições (C++11).....	359
15.3.2.1 • Inicialização uniforme (C++11).....	362
15.3.3 • Resultados de operações.....	363
15.3.3.1 • Operações relacionais.....	363
15.3.3.2 • Operações lógicas.....	363
15.3.3.3 • Operações aritméticas.....	363
15.3.4 • Posição dos operadores de incremento e decremento.....	364
15.3.5 • Operadores <i>bit a bit</i> .....	364
15.3.5.1 • AND.....	365
15.3.5.2 • OR.....	365
15.3.5.3 • NOT (ou complemento de 1).....	365
15.3.5.4 • XOR ( ou exclusivo).....	366
15.3.5.5 • Shift (ou deslocamento) à esquerda.....	366
15.3.5.6 • Shift (ou deslocamento) à direita.....	366
15.3.5.7 • Exemplos com operadores de <i>bits</i> .....	367
15.3.5.7.a • Exemplo com <i>and</i> , <i>or</i> , e <i>not</i> .....	367
15.3.5.7.b • Exemplo com <i>shift</i> (à esquerda e à direita).....	369
15.3.5.7.c • Exemplo com <i>xor</i> (1).....	370
15.3.5.7.d • Exemplo com <i>xor</i> (2).....	372
15.4 • Controles do fluxo de processamento.....	375
15.4.1.1 • Parâmetros para funções e retorno de funções.....	375
15.4.1.2 • Declarando parâmetros.....	375
15.4.1.3 • Formas de declaração.....	375
15.4.1.3.a • Forma de declaração válida em em C e C++.....	375
15.4.1.3.b • Forma de declaração inválida em C++.....	375
15.4.1.4 • Funções com quantidade fixa de parâmetros.....	375
15.4.1.5 • Funções com quantidade variável de parâmetros.....	376
15.4.1.5.a • Declarando os parâmetros desconhecidos.....	376
15.4.1.5.b • Exemplo de função com quantidade de parâmetros variável.....	377
15.4.1.6 • Valor de retorno, tipo e protótipo de funções.....	377
15.4.1.6.a • Valor de retorno.....	377
15.4.1.6.b • Tipo de uma função.....	378
15.4.1.6.c • Protótipo de funções.....	379
15.4.1.7 • Modos de passagem de parâmetros (recapitulando).....	380
15.4.1.7.a • 1 - Passando parâmetros por valor.....	380

15.4.1.7.b	2 - Passando parâmetros por endereço.....	381
15.4.1.7.c	3 - Passando parâmetros por referência.....	384
15.4.1.8	Há dois motivos para passar parâmetros por endereço ou por referência. .....	385
15.4.1.9	Impedindo efeitos colaterais desnecessários.....	386
15.4.2	Lógica estruturada através de objetos.....	386
15.4.3	Controles de laço (recapitulando).....	387
15.4.3.1	O laço <i>for</i> .....	387
15.4.3.1.a	Forma geral e funcionamento.....	387
15.4.3.1.b	Sintaxe.....	388
15.4.3.1.c	Cuidados a tomar com o laço <i>for</i> .....	390
15.4.3.2	O laço <i>while</i> .....	391
15.4.3.2.a	Forma geral e funcionamento.....	391
15.4.3.2.b	Sintaxe.....	391
15.4.3.2.c	Cuidados a tomar com o laço <i>while</i> .....	392
15.4.3.3	Considerações gerais sobre os laços <i>for</i> e <i>while</i> ( <i>tenha cuidado com...</i> ) .....	394
15.4.3.4	O laço <i>do ... while</i> .....	394
15.4.3.4.a	Forma geral, funcionamento e sintaxe:.....	394
15.5	Ponteiros, Matrizes e Referências.....	397
15.5.1.1	Alocando livremente memória no <i>heap</i> (evite...).....	397
15.5.1.2	Acessando valores através de ponteiros.....	398
15.5.1.3	Exercícios: demonstrando o funcionamento de ponteiros.....	398
15.5.2	Ponteiros inteligentes (smart pointers).....	401
15.5.2.1	Auto ptr (obsoleto a partir de C++11).....	403
15.5.2.2	unique_ptr (C++11).....	403
15.5.2.3	shared_ptr (C++11).....	404
15.5.2.4	weak_ptr (C++11).....	405
15.5.3	Vetores e Matrizes.....	406
15.5.3.1	Onde alocar vetores: memória global, pilha ou <i>heap</i> ?.....	407
15.5.3.2	Matrizes - usando múltiplas dimensões.....	409
15.5.3.3	Inicialização de vetores e matrizes.....	410
15.5.3.4	Vetores e Matrizes de estruturas.....	410
15.5.3.4.a	Inicializando uma matriz de estruturas.....	411
15.5.3.5	Vetores e Matrizes de caracteres.....	412
15.5.3.5.a	Inicializando uma matriz de caracteres.....	413
15.5.3.5.b	Entendendo matriz de caracteres como um endereço.....	413
15.5.3.5.c	Exemplo de matriz de caracteres.....	414
15.6	Ponteiros para funções(exemplo adicional).....	416
	A estrutura relatório.....	416

## 15.1 • Conversões de tipos

### 15.1.1 • Conversões no estilo C

Quando somamos um dado do tipo `int` e um dado do tipo `char`, ou um dado do tipo `int` e um dado do tipo `float`, C++ converte o tipo menos preciso para o tipo mais preciso presente naquela operação.

- É uma conversão temporária, isto é, feita apenas para fins da operação.
- Desse modo na soma de um `int` com um `float`, o `int` (menos preciso) seria provisoriamente convertido para `float` (mais preciso) de modo a viabilizar a operação sem perda de precisão.

Contudo, pode ser necessária uma **conversão explícita para fins de operação**.

- Considere a seguinte situação:
  - uma variável `float` deverá receber (atribuição) o resultado da soma de outro `float` mais o resultado da divisão entre dois `int`'s. [ **`f1 = f2 + ( i1 / i2 )`** ]
  - neste caso, **se** a divisão entre as duas variáveis do tipo `int` **resultar** em um número **real** (isto é contendo parte fracionária ou casas decimais), a precisão será perdida, pois a parte fracionária é **eliminada** numa operação entre tipos que não a comportam.

*Exemplos:*

- **4/2 - o resultado é 2; não há perda de precisão;**
- **5/2 - o resultado é 2.5; há perda de precisão,**



como a parte fracionária é eliminada, o resultado será **truncado** para 2.

- O problema não teria ocorrido, logicamente, se todos os dados envolvidos na operação fossem do tipo `float`.
- Contudo, é possível em C++ superar o problema de outro modo:



através da **conversão explícita** para fins da operação.

- Para isso utilizamos os **moldes de tipo**:
- um molde de tipo é o nome do tipo entre parênteses.
  - Desse modo, na operação:

```
int i1, i2;
```

```
.....
```

```
float f2 = f + ( (float)i1 / i2 ) // admitido em C e C++
```

**OU**

```
f2 = f + ( float( i1 ) / i2 ) // apenas em C++
```



Não há perda de precisão, **pois** o tipo do resultado da operação assume o tipo do **operando mais preciso** (então, bastou elevar a precisão do primeiro operando).

*Exemplo:*

```
int main( )
{
    int A = 5 ;
    char C = 2 ;
    float F = 10.0 ;
    int A2 ;
```



```

float F2 ;
A2 = A + C ;
cout << "int + char = " << A2 << endl ; // resultado : 7

F2 = A + C + F ;
cout << "int + char + float = " << F2 << endl ; // resultado: 17

F2 = F + ( A / C ); // ➡ não usa conversão explícita
cout << "float + (int / char ) = " << F2 << endl ;
// resultado (10 + 2.5) = 12; ➡ houve perda de precisão

F2 = F + ( float( A ) / C ); // ➡ aqui, usa conversão
cout << "float + (int / char ) = " << F2 << endl ;
// resultado (10 + 2.5) = 12.5; ➡ não houve
// perda de precisão

return 0;
}

```

Além disso, C++ oferece palavras reservadas especiais para casts mais explícitos.

### 15.1.2 • Conversões no estilo C++

Como já vimos acima, o C++ admite o estilo "C" de conversão entre tipos.

Por exemplo:

```

float x;
.....
int a = (int)(x / 3);

```

Uma variação do *cast* tradicional do C, exemplificado acima, seria o *cast* no formato função, admitido por C++:

```
int a = int( x / 3);
```

Em muitos casos um *cast* pode representar um problema potencial.

Por exemplo:

```
int * pa = (int *) 1000; // "1000" é um endereço VÁLIDO?
```

Ou então:

```

const int * pi = ..... ;
.....
int * p = (int *)pi; // transformou um ponteiro const em não-const

```

Para dar mais segurança e legibilidade aos *cast's* o C++ introduziu alternativas específicas para *cast*, cada qual expressando o problema potencial daquele tipo de *cast*.

São elas:

- `static_cast`;
- `dynamic_cast`;
- `reinterpret_cast`;
- `const_cast`;

#### a. `static_cast`

O *static\_cast* é usado para converter um ponteiro para uma classe-base para um ponteiro para uma classe derivada, ou vice-versa

**Exemplo:**

```
class Base {};
```

```

class Deriv : public Base {;
void Funcao ( Base * pB, Deriv * pD )
{
    // 1) convertendo ponteiro a objeto-base para derivado:
    Deriv * pDTemp = static_cast< Deriv * > (pB);

    // 2) convertendo ponteiro ao objeto-derivado para base:
    Base * pBTemp = static_cast < Base * > (pD);
}

```

Observar que no primeiro caso convertemos um ponteiro para a classe base sendo convertido para um objeto derivado.

Se, contudo, isso não for verdadeiro (isto é, se o objeto for na verdade um objeto-base) e tentarmos acessar membros derivados ocorrerá erro já que esses membros não estarão alocados na memória.

Além disso o *static\_cast* pode ser usado para as conversões de tipos básicos relacionados (inteiros, por exemplo):

```

int i;
.....
char c = static_cast<char> (i);

```

#### b. dynamic cast

O *static\_cast* feito no primeiro caso acima (conversão de ponteiro a derivada para base) não é seguro.

Se ele fosse realmente necessário, poderíamos então usar *dynamic\_cast* que permitiria uma **checagem em tempo real**.

```

Deriv * pDTemp = dynamic_cast< Deriv * > (pB);

```

Caso **pB** não aponte para um objeto do tipo **Deriv**, teríamos um **retorno nulo**.

Lembrar contudo que este recurso tem um **custo** mais alto, pois a verificação estará sendo feita durante a própria execução.

Assim, se quisermos usá-lo deveremos **habilitar** a opção do compilador para **suporte** a "RTTI" (*Run-Time Type Information*).

#### c. reinterpret\_cast

As conversões mais suspeitas devem ser feitas com *reinterpret\_cast*, que pode ser usado para converter um ponteiro de um tipo para qualquer outro e inclusive um inteiro para um ponteiro:

```

char * pC;
.....
int * pI = reinterpret_cast < int * > (pC);
pI = reinterpret_cast < int *> (100);

```

#### d. const\_cast

Geralmente, conversões de *const* para não-*const* são altamente suspeitas e devem ser documentadas e facilmente localizáveis com o uso deste conversor.

```
const char * pC;
.....
char * pC2 = const_cast < char *>(pC);
```

## 15.2 • Tipos agregados e aninhados

As diretivas **struct** e **class** cumprem um papel muito mais amplo do que o **enum** já que, através delas, podemos criar conjuntos de dados (de quaisquer tipos já existentes) e funções.


Como já vimos, os dados podem ser protegidos através de uma restrição de acesso que os torna privativos da estrutura (isto é, só poderão ser acessados nas funções declaradas no interior da própria estrutura. Para isso, usa-se a palavra reservada **private**.

Em contrapartida, podemos declarar membros (dados ou funções) sem restrição de acesso, classificando-os como **públicos**. Para isso usa-se a palavra reservada **public**.

Apenas as funções membro de uma *struct/class* (e como veremos, as suas “amigas”) podem acessar os membros *private*.

Aqui, o que é necessário registrar é que, além dos tipos primitivos oferecidos pela linguagem, e dos tipos simples que o programador pode criar com **enum**, a linguagem permite também a **criação de tipos complexos**, isto é que permitem **reaproveitar e re-combinar** todos os tipos criados anteriormente.

Por exemplo, mais acima criamos o tipo “*Data*”, reaproveitando, em uma combinação específica, os tipos primitivos **char**, **short** e **bool**.

 E, agora, podemos **reaproveitar** o próprio tipo “*Data*”, em um novo tipo, ainda mais complexo:

```
class Contas
{
    public:
        enum TipoConta { Receita , Despesa }; // enum aninhado em “Contas”
    private:
        double m_dblValor ;
        TipoConta m_TipoConta ;
        Data m_dtEntrada ; // class Data agregada em “Contas
        Data m_dtVencimento ; // idem
        .....
    public:
        .....
};
```

O tipo “*Contas*”, reaproveita os tipos primitivos **double** e **char**.

Também cria e agrega, através de **enum**, um novo tipo simples (“*TipoConta*”), para melhor expressar a natureza de um valor informado.

E **reaproveita o tipo estruturado “Data”**, agregando-o duas vezes ao conjunto de dados (data de entrada e data de vencimento).

### 15.2.1 ▪ Tipos aninhados .

Quando escrevemos uma estrutura como “Data”, sabemos que ela será reaproveitada em um grande número de outras estruturas.

Ou seja: esse tipo não foi criado para servir **apenas** à estrutura “Contas”, **mas também** a muitas outras que serão oportunamente criadas.

Mas às vezes podemos precisar de um tipo **auxiliar** que só será usado em uma **única** estrutura.

Observe que isso já foi feito acima , na estrutura “Contas”, quando foi criado o tipo “TipoConta” através de **enum**.

Caso esse tipo de dado fosse usado também em outras estruturas, então ele deveria ter sido criado **fora** da estrutura “Contas”, para ser aproveitado nas diversas estruturas que dele necessitassem.

Como não era esse o caso (já que só seria usado na própria estrutura “Contas”), então ele foi criado **aninhado** na estrutura “Contas”.

Assim, ele é um **tipo interno** à própria estrutura. Isto significa que não só a **variável** *m\_TipoConta* é um membro de “Contas”. **O próprio tipo** (“TipoConta”) **pertence à estrutura**.

Assim, se fossemos usar esse tipo em qualquer outro lugar que não a estrutura “Contas”, teríamos que respeitar essa realidade, da seguinte maneira:

```
Contas::TipoConta Variavel ; // usando fora da classe
Variavel = Contas::Receita ; // idem
```

### 15.2.2 ▪ Estruturas aninhadas.

Podemos criar um tipo aninhado em uma estrutura. Seja usando um **enum dentro** da estrutura, ou também podemos criar um tipo aninhado utilizando **struct** ou **class**. Isto é: podemos criar uma **estrutura aninhada em outra**.

Fazemos isso sempre que precisamos de um tipo estruturado auxiliar para uma estrutura maior. E esse tipo **não será** reaproveitado em outras estruturas.

**Exemplo:**

```
struct Contas
{
    .....
private:
    struct ImpostoSimples // tipo criado com restrição de acesso private
    {
        public:
            double m_dbIPercentual ;
            Data m_dtPagamento ;
            .....
    } ;
public:
    struct ImpostoPorFaixas // tipo criado sem restrições de acesso
    {
        public:
```

```

        double m_dblPercentual ;
        double m_dblValorMinimo ;
        double m_dblValorMaximo ;
        .....
    } ;

    // cria dois membros de dados a partir do tipo interno
    ImpostoSimples m_isISS ;
    ImpostoSimples m_isINSS ;
    .....
} ;

```

Um dos tipos auxiliares internos (*ImpostoSimples*) foi criado com restrição de acesso **private**. Então **não** poderá ser usado **fora** da classe.

Já o tipo *ImpostoPorFaixas* é público (**public**) e poderia ser usado da seguinte forma:

```
Contas::ImpostoPorFaixas Variavel ; // usando fora da classe
```

Isto é: o nome do tipo é **Contas::ImpostoPorFaixas** e **não** *ImpostoPorFaixas*, justamente porque é um tipo **aninhado**, logo **pertence ao ambiente de nomes** da estrutura **mais externa**.

### 15.2.2.1 • Tipos aninhados: em C eles são apenas indicativos.

Em C, a declaração de uma variável de um tipo aninhado seria:

```
ImpostoPorFaixas Variavel ;
```

Desse modo, o tipo **ImpostoPorFaixas**, embora declarado dentro do tipo **Contas**, **não pertence a Contas**.

### 15.2.2.2 • Tipos aninhados: C++ não permite o estilo C.

Um outro ponto em que C++ não é compatível com C é esse.

Em C++, como já vimos acima, isso **não é permitido**, e o programador terá que alterar esse código, caso escrito anteriormente para compiladores C, para que seja compatível com a única escrita admitida em C++:

```
Contas::ImpostoPorFaixas Variavel ;
```



Em **C++**, portanto só é possível a utilização da forma indicada acima, a qual implementa a lógica do encapsulamento: *ImpostoPorFaixa* é um tipo que pertence ao tipo *Contas*.

### 15.2.2.3 • Criação de tipos diretamente no retorno ou nos parâmetros de uma função.

Em C podemos criar um tipo diretamente no retorno ou na declaração de parâmetros de uma função:

```

struct Area { int Largura ; int Altura; } Calcula();
/* A função Calcula retorna um valor do tipo "Area" declarado diretamente na especificação
do tipo de retorno da função */

Calcula( struct Area { int Largura ; int Altura; } AreaACalcular );
/* A função Calcula recebe o parâmetro "AreaACalcular" cujo tipo ("Area") é declarado
diretamente na especificação do tipo do parâmetro da função */

```

Em C++ isso **não é permitido**. Assim, qualquer código herdado do C que contenha esse tipo de escrita deverá ser reescrito. Observar que alguns compiladores ainda permitem essa escrita, mas ela não é padrão.

## 15.3 • Criando tipos através de *union*.

Unões funcionam como estruturas polimórficas.

Isto significa que uma união pode abrigar **diferentes tipos** mas apenas um **apenas um** poderá ser usado **a cada vez**.

Isto é, o tamanho de uma *union* é o tamanho do maior tipo declarado em seu interior. Os campos de tamanho menor, quando usados, simplesmente não preenchem todo o espaço reservado, deixando uma área de memória vazia, sem uso.

Precisamos sempre saber qual é o campo da *union* que está em uso no momento para não recuperar informações incorretas.

### **Exemplo:**

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

#define VT_DOUBLE 1
#define VT_BOOL 2
#define VT_LONG 3
#define VT_CHAR 4

struct VariosTipos
{
    int TipoEmUso;
    union
    {
        double dblVal;
        bool boolVal;
        long longVal;
        char charVal;
    };
    void ImprimeVariosTipos( );
};

void VariosTipos::ImprimeVariosTipos( )
{
    switch ( TipoEmUso )
    {
        case VT_DOUBLE:
            cout << dblVal << endl; return;
        case VT_BOOL:
            cout << boolVal << endl; return;
        case VT_LONG:
            cout << longVal << endl; return;
        case VT_CHAR:
            cout << charVal << endl; return;
        default:
            cout << "ERRO" << endl;
    }
}

int main()
{
```

```

VariosTipos vrt1, vrt2;

vrt1.TipoEmUso = VT_DOUBLE;
vrt1.dblVal = 20.5;
vrt1.ImprimeVariosTipos ();

vrt1.TipoEmUso = VT_LONG;
vrt1.longVal = 1050L;
vrt1.ImprimeVariosTipos ();

vrt2.longVal = 3150L;
vrt2.ImprimeVariosTipos ();

return 0;
}

/*
    RESULTADO:
        20.5
        1050
        ERRO
*/

```

A melhor maneira de utilizar uniões é definindo, fora da **union**, uma variável indicadora para descobrir qual é o tipo que está sendo efetivamente usado.

No exemplo acima, isso foi feito aninhando a **union** em uma **struct** e incluindo um membro de dados na **struct** para indicar qual é o tipo atualmente em uso na **union**.

Em C, nas situações onde é preciso trabalhar com uma variedade alternativa de tipos, a **union** é quase sempre a primeira possibilidade a ser considerada (uma outra saída para o problema seriam os ponteiros **void**).

Já em C++, antes de lançar mão de uma **union**, devemos primeiro analisar se não poderia ser empregado um **template** (que é justamente o assunto do próximo capítulo).

E apenas em caso negativo é que devemos usar a **union**.

### 15.3.1 ▪ Regras para *union*, específicas de C++.

Em uma *union* C++, aplicam-se as seguintes **permissões**:

- a *union* pode especificar diferentes **acessos** (*public*, *private*, ...), sendo que o acesso **default** é *public*;
- pode conter tanto membros de **dados** como **funções**-membro;
- a afirmação acima aplica-se, inclusive, a construtoras e destrutoras: uma *union* C++ pode conter **construtoras** e **destrutora**;

E, em uma *union* C++, aplicam-se também as seguintes **restrições**:

- a *union* **não pode** conter funções **virtuais** ou membros **estáticos** (*veremos mais a frente o que são membros estáticos e funções virtuais*);
- uma *union* **não pode** ser usada para fins de **herança**, pois não pode ser usada como base ou como derivada de uma base (*veremos mais a frente o que é herança, o que é uma classe base e o que é uma classe derivada*).

### 15.3.2 ▪ Unions com menos restrições (C++11)

Muitas das restrições das unions foram removidas no padrão C++11, tornando-as mais flexíveis. As unions agora:

- Podem definir funções membros especiais.
- Podem acessar objetos com funções membro não triviais
- Podem definir membros com acesso non-public.

Embora tenha havido uma certa flexibilização, algumas restrições ainda permanecem para as unions:

- Não podem declarar referencias para variáveis.
- Não podem possuir funções virtuais.
- Não podem ser usadas como base ou derivada em herança.

```
#include <iostream>

using namespace std;

union UnionTest
{
    int n;
    //Apaga a destrutora e a construtora implicitamente
    string s;

    UnionTest()
    {
        cout << "UnionTest()\n";
    }

    ~UnionTest()
    {
        cout << "~UnionTest()\n";
    }

    void setX(double value)
    {
        x = value;
    }

    double getX()
    {
        return x;
    }
private:
    double x;
};

int main()
{
    UnionTest t;
    t.n = 10;
    cout << t.n << "\n\n";
    //Placement new - sobrecarga do operador new para instanciar minha string.
    //Com isso a string s passa a ser um membro ativo da union.
    new (&t.s) basic_string<char>("Agit");
    cout << t.s << "\n\n";
}
```



```
//Destrói a string
t.s.~basic_string<char>();
t.setX(99.8);
cout << t.getX() << "\n\n";
return 0;
}
```

### 15.3.2.1 ▪ Inicialização uniforme (C++11)

No C++, podemos inicializar variáveis de várias maneiras diferentes:

```
class A
{
    int a;
public:
    A(int n) : a(n) {}
};

struct B {
    int a,b;
}

int main()
{
    int a=1;           // Inicialização de tipos básicos com o sinal de igual (=)
    int b(1);          // Inicialização de tipos básicos com parênteses
    A meuA(10);        // Inicialização de membros de classes
    B meuB = {10,20};  // Inicialização de agregados
}
```

Havia algumas limitações e restrições para se inicializar alguns tipos em determinadas situações no padrão C++03:

- 1) Não havia uma maneira para inicializar vetores membros de classe;
- 2) Não havia uma maneira de inicializar vetores alocados no heap;
- 3) Não havia como inicializar objetos das classes containers da STL

No C++11, ficou definido que a inicialização deveria ser feita de uma maneira única, através do uso das chaves {} da seguinte forma:

```
class Teste
{
    int vetor[3];
public:
    Teste() : vetor{0, 1, 2} // Inicialização de vetores membros de classe
    {
    }
};

int main()
{
    int a {1};           // Inicialização de tipos básicos com o sinal de igual (=)
    A meuA{10};          // Inicialização de membros de classes
    B meuB {10,20};      // Inicialização de agregados
    int *vetor = new int[3]{0, 1, 2}; // Inicialização de vetores alocados no heap
    // Inicialização de containers da STL
    vector<string> nomes({"Pedro", "Paulo", "Jose"});
}
```

### 15.3.3 ▪ Resultados de operações.

Toda operação deve retornar um determinado **resultado**. Vejamos como isso ocorre com os diferentes tipos de operadores.

#### 15.3.3.1 ▪ Operações relacionais.

Os operadores relacionais permitem estabelecer uma **relação comparativa** entre dois operandos.

Como ocorre em toda e qualquer operação, uma operação relacional deverá retornar um resultado.

E o resultado de uma operação relacional é sempre **zero**(falso) ou **um**(verdadeiro). Isto significa que esse tipo de operação retorna sempre um resultado **lógico**.

*Assim, se fizermos:*

```
int a ;  
.....  
int z = ( a > 5 ) ; // O valor de "a" está maior do que 5 ?
```

*O resultado será:*

- Se "**a**" estiver **maior** do que **5**, o resultado da operação será **1** (um, verdadeiro); e, em seguida, esse valor será armazenado em "**z**".
- Se "**a**" **não** estiver **maior** do que **5** (ou seja, se estiver menor ou igual), o resultado da operação será **0**(zero, falso); e, em seguida, esse valor será armazenado em "**z**".

#### 15.3.3.2 ▪ Operações lógicas.

Também aqui o **resultado** só poderá ser **lógico**.

*Assim, se fizermos:*

```
int a , b ;  
.....  
int z = ( a && b ) ; // "a" está verdadeira (diferente de zero) E(&&)  
// "b" também está verdadeira (diferente de zero)?
```

*O resultado será:*

- Se "**a**" estiver **diferente de zero** e "**b**" também estiver **diferente de zero**, o resultado da operação será **1** (um, verdadeiro); e, em seguida, esse valor será armazenado em "**z**".
- Se um dos dois ("**a**" ou "**b**") estiver **igual a zero**, o resultado da operação será **0** (zero, falso); e, em seguida, esse valor será armazenado em "**z**".

#### 15.3.3.3 ▪ Operações aritméticas.

Neste caso o resultado será, naturalmente, **aritmético**.

```
int a , b = 10 , c = 20 ;  
  
a = b + c ; // o resultado da soma de "b" e "c" é o número inteiro 30;  
// esse resultado (aritmético) é então armazenado em "a".
```

### 15.3.4 ▪ Posição dos operadores de incremento e decremento.

Os operadores de incremento(++ ) e decremento(-- ) admitem duas situações:

- a. Posicionamento **pósfixado**: o operador aparece **depois** (à direita) do operando.

*Exemplos:*

```
1 ) a++ ;
2 ) b = a++;
```

Esse tipo de posicionamento apresenta o seguinte comportamento: se tentarmos **ler** o valor da variável **a**, ele será devolvido **antes** que a operação de incremento seja executada. **Assim:**

```
a = 7 ;
b = a++ ;
```

Após a operação de incremento, o valor de **b** será **7**. E o valor de **a** será **8**.

- Isto porque **primeiro** a variável **a** foi **lida**, retornando **7**, para **só depois** ser incrementada para **8**.

- b. Posicionamento **préfixado**: o operador aparece **antes** (à esquerda) do operando.

*Exemplos:*

```
1 ) ++a ;
2 ) b = ++a ;
```

Esse tipo de posicionamento apresenta o seguinte comportamento: se tentarmos **ler** o valor da variável **a**, ele será devolvido **após** a execução da operação de incremento.

**Assim:**

```
a = 7 ;
b = ++a ;
```

O valor de **b** será **8**. E o valor de **a** será **8**.

Isto porque **primeiro** a variável **a** foi **incrementada**, assumindo o valor **8**, e **só depois** foi **lida**, retornando o valor **8**.

---

 *Em resumo:*

---

- Quando escrevemos **x++** (variável antes, incremento depois) estamos ordenando:
  - **Primeiro leia o que está armazenado na variável 'x' e depois incremente 'x'.**
- E quando escrevemos **++x** (incremento antes, variável depois) estamos ordenando:
  - **Primeiro incremente a variável 'x' e depois leia o que está armazenado lá.**
- E o mesmo é válido para o operador de **decremento (--)**.

### 15.3.5 ▪ Operadores *bit a bit*.

*Regra básica:*



operadores de bit's só podem ser usados com tipos **inteiros**, **não** sendo suportados em tipos com ponto flutuante.

Já vimos, na primeira parte da apostila, o operador **and** em operações **bit a bit**. Vejamos agora todos os operadores **bit a bit**:

- **and, or, not, xor, shift** à esquerda e **shift** à direita.

### 15.3.5.1 • AND

Considerando:

```
unsigned char A = 8 ;
unsigned char B = 9 ;
```

Podemos fazer:

```
A = A & B ; // and de bits
```

```
A &= B ; // Melhor, pois, como "A" aparece nos dois lados da operação,
// podemos utilizar o operador composto.
```

Variável	Decimal	Binário
A	8	0000 1000
B	9	0000 1001
A (Resultado)	8	0000 1000

Uma operação **and** exige que os dois operandos (neste caso, **cada bit**) estejam verdadeiros.

Assim, apenas se **ambos** os *bits* estiverem com **um** (verdadeiros) o resultado será **verdadeiro** (um).

Do contrário o resultado será **falso** (zero).

### 15.3.5.2 • OR

Considerando:

```
unsigned char A = 8 ;
unsigned char B = 9 ;
```

Podemos fazer:

```
A = A | B ; // or de bit's.
```

```
A |= B ; // Melhor, pois, como "A" aparece nos dois lados da operação,
// podemos utilizar o operador composto
```

Variável	Decimal	Binário
A	8	0000 1000
B	9	0000 1001
A (Resultado)	9	0000 1001

Em uma operação **or** basta que um dos dois operandos (neste caso, **cada bit**) esteja verdadeiro.

Assim, se qualquer um dos dois *bits* envolvidos na operação *bit a bit* estiver com **um** (verdadeiro) o resultado será **verdadeiro** (um). Do contrário o resultado será **falso** (zero).

### 15.3.5.3 • NOT (ou complemento de 1)

Considerando:

```
unsigned char A = 8;
```

Podemos fazer:

```
A = ~A ; // not de bit's.
```

Variável	Decimal	Binário
A	8	0000 1000
A (Resultado)	247	1111 0111

Uma operação **not bit a bit**, **inverte** todos os *bits*. (zero é trocado para um, e um é trocado para zero).

Observe que o **not** aplicado sobre o número 8 (exemplo acima) resulta no número **247**. E o mesmo resultado poderia ser obtido com **255 – 8** (o que será válido para qualquer operação **not** de *bits* em cada *byte*).

Sabendo-se que 255 tem todos os seus *bits* com **um** (1111 1111), podemos dizer que uma operação **not bit a bit** é o mesmo que o **complemento de um**.

### 15.3.5.4 • XOR ( or exclusivo)

Considerando:

```
unsigned char A = 8 ;
unsigned char B = 9 ;
```

Podemos fazer:

```
A = A ^ B ; // xor de bit's.
A ^= B ;    // Melhor, pois, como "A" aparece nos dois lados da operação,
            // podemos utilizar o operador composto
```

Variável	Decimal	Binário
A	8	0000 1000
B	9	0000 1001
A (Resultado)	1	0000 0001

Em uma operação **xor** o resultado será verdadeiro apenas se os dois operandos (os dois *bits*) estiverem **diferentes** (“*exclusivamente OU*”).

Isto significa que **um deles** deve estar **verdadeiro** e o **outro** deve estar **falso**.

### 15.3.5.5 • Shift (ou deslocamento) à esquerda

Considerando:

```
unsigned char A = 8;
```

Podemos fazer:

```
A = A << 1 ; // shift de bit's, com deslocamento de um bit
            // para a esquerda;
A <<= 1;     // Melhor, pois, como "A" aparece nos dois lados da operação,
            // podemos utilizar o operador composto
```

Variável	Decimal	Binário
A	8	0000 1000
A (Resultado)	16	0001 0000

O segundo operando (1, no exemplo), indica a **quantidade de bits** que deve ser deslocada **à esquerda** no primeiro operando. Os bit's vagos, à direita, são preenchidos com zeros (o primeiro bit, no exemplo).

### 15.3.5.6 • Shift (ou deslocamento) à direita:

Considerando:

```
unsigned char A = 8;
```

Podemos fazer:

```

A = A >> 1 ; // shift de bits, com deslocamento de um bit
               // para a direita;
A >>= 1;    // Melhor, pois, como "A" aparece nos dois lados da operação,
               // podemos utilizar o operador composto

```

Variável	Decimal	Binário
A	8	0000 1000
A (Resultado)	4	0000 0100

O segundo operando (**1**, no exemplo), indica a **quantidade de bits** que deve ser deslocada **à direita** no primeiro operando. Os *bits* vagos, à esquerda, são preenchidos com zeros (o último bit, no exemplo).

### 15.3.5.7 ▪ Exemplos com operadores de *bits*.

#### 15.3.5.7.a ▪ Exemplo com *and*, *or*, e *not*.

Um bom exemplo para uso de *bits* é quando precisamos de um conjunto de variáveis para indicar estados verdadeiro/falso.

Ao invés de criar diversas variáveis separadas podemos usar os *bits* de uma única variável (pois para indicar se uma situação está verdadeira ou falsa basta usar um único bit, que poderá estar com um(verdadeiro) ou com zero(falso).

Assim poderíamos ter uma situação de entrada de dados onde diversos campos precisavam ser preenchidos. E usariamos os *bits* de uma variável para indicar a situação de cada um dos campos: se preenchido corretamente ou não.

**Vamos então criar uma classe chamada *Dados*.**

- A classe **Dados** tem três atributos: *Código*, *Valor* e *Tipo*.
- Além disso terá uma variável interna chamada "*Erros*" que servirá para assinalar qualquer erro.

**Regras:**

- *Código* deverá estar entre as constantes *CODIGO\_MIN* e *CODIGO\_MAX*.
- *Valor* deverá ser maior que zero.
- E *Tipo* deverá assumir um destes dois valores constantes: *TIPO\_RECEITAS* ou *TIPO\_DESPESAS*

**Funcionamento:**

- Cada um dos três campos acima, usará um *bit* da variável *Erros*, para indicar se foi preenchido corretamente.
- Se qualquer um dos três campos estiver **incorreto, um bit da variável *Erros*, correspondente ao campo, deverá ser ligado.**
- E, se estiver **correto** o bit deverá ser **desligado**.
- Para que isso seja possível, é preciso que cada campo use um **único bit**. Para isso é preciso que **o indicador de erro de cada campo seja uma potência de 2** ( 1, 2, 4, 8, 16, etc).

**Implementação:**

```

#include <iostream>
class Dados
{
    private:

```

```

int m_Codigo;
double m_Valor;
int m_Tipo;
unsigned int m_Erros;

public:
    // Constantes indicadoras de erro em um campo:
    // cada uma é uma potência de 2
    enum { ERRO_CODIGO = 1,    // 2 elevado a zero
          ERRO_VALOR  = 2,    // 2 elevado a um
          ERRO_TIPO   = 4 };  // 2 elevado a dois

```

/\* E COMO usar ?

1) Se quisermos assinalar o bit correspondente ao campo *Codigo* como estando **ERRADO**:

```
m_Erros |= ERRO_CODIGO; // coloca este bit com 1 usando or de bits
```

2) E se quisermos zerar esse bit, assinalando-o como correto, devemos inverter (not) todos os bits do indicador *ERRO\_CODIGO* e, então, aplicar um **and** de bits:

```
m_Erros &= ~ERRO_CODIGO; // coloca este bit com zero
```

3) E se quisermos saber se o bit de *Codigo* está ligado:

```
if ( m_Erros & ERRO_CODIGO ) // basta usar o and de bits
```

```
.....
```

\*/

```
// Outras constantes
```

```
enum { CODIGO_MIN = 100, CODIGO_MAX = 999 };
enum { TIPO_RECEITAS = 1, TIPO_DESPESAS = 2 };

```

```
Dados() // construtora
```

```
void AlterarCodigo(int Cod);
```

```
void AlterarTipo(int Tipo);
```

```
void AlterarValor(double Valor);
```

```
int LerCodigo() const { return m_Codigo; }
```

```
double LerValor() const { return m_Valor; }
```

```
int LerTipo() const { return m_Tipo; }
```

```
int LerErros() const { return m_Erros; }
```

```
};
```

```
Dados::Dados() // construtora
```

```
{
```

```
    m_Erros = 0;
```

```
    AlterarCodigo(0);
```

```
    AlterarValor(0);
```

```
    AlterarTipo(0);
```

```
}
```

```
void Dados::AlterarCodigo(int Cod)
```

```
{
```

```
    m_Codigo = Cod;
```

```
    if ( Cod >= CODIGO_MIN && Cod <= CODIGO_MAX )
```

```
        m_Erros &= ~ERRO_CODIGO; // Código CORRETO:
```

```
        // então coloca este bit com zero
```

```
    else
```

```
        m_Erros |= ERRO_CODIGO; // Código ERRADO: então coloca este bit com 1
```

```
}
```

```
void Dados::AlterarValor(double Valor)
```

```
{
```

```
    m_Valor = Valor;
```



```

    if ( Valor > 0 )
        m_Erros &= ~ERRO_VALOR ; // Valor CORRETO: então coloca este bit com zero
    else
        m_Erros |= ERRO_VALOR ; // Valor ERRADO: então coloca este bit com 1
}

void Dados::AlterarTipo(int Tipo)
{
    m_Tipo = Tipo;
    if ( Tipo == TIPO_RECEITAS || Tipo == TIPO_DESPESAS )
        m_Erros &= ~ERRO_TIPO ; // O tipo está CORRETO:
                                // então coloca este bit com zero
    else
        m_Erros |= ERRO_TIPO ; // O tipo está ERRADO: coloca este bit com 1
}

```

---

```

int main()
{
    Dados MeusDados;
    MeusDados.alterarValor(0); // preenchimento incorreto...
    MeusDados.alterarCodigo (1); // preenchimento incorreto...
    MeusDados.alterarTipo(TIPO_RECEITAS); // preenchimento
                                         // CORRETO.

    int Erros = MeusDados.LerErros();
    if ( Erros == 0 )
        cout << "Nenhum erro !!!" << endl;
    else
    {
        if ( Erros & ERRO_CODIGO ) // and de bits
            cout << "Codigo esta errado" << endl;
        if ( Erros & ERRO_VALOR ) // and de bits
            cout << "Valor esta errado" << endl;
        if ( Erros & ERRO_TIPO ) // and de bits
            cout << "Tipo esta errado" << endl;
    }
    return 0;
}
/* RESULTADO

    Codigo esta errado
    Valor esta errado
*/

```

#### 15.3.5.7.b • Exemplo com *shift* (à esquerda e à direita).

Vamos agora ver um segundo exemplo, usando o **shift** para a esquerda e para a direita. O objetivo é “Imprimir uma lista crescente de números, começando em “2 elevado a zero” (1) até atingir “2 elevado a 5” (32). Em seguida imprima em ordem inversa (de-crescente), até atingir o número zero.

Devem ser usados os operadores **shift**”.

```

main()
{
    int Var ;
    // shift para a esquerda: 

```

```

for ( Var = 1 ; Var <= 16 ; Var <<= 1 ) // shift à esquerda:
                                         // lista crescente

    cout << Var << " ";

cout << "\nValor de Var depois do primeiro for = " << Var
    << endl << endl;

// shift para a direita:—————▼
for (      ; Var >= 1 ; Var >>= 1 ) // shift à direita:
                                         // lista decrecente

    cout << Var << " ";

cout << endl;
cout << "Valor de Var depois do segundo for = " << Var << endl ;
return 0;
}

/* RESULTADO
    1  2  4  8 16
    Valor de Var depois do primeiro for = 32
    32 16  8  4  2  1
    Valor de Var depois do segundo for = 0
*/

```

### 15.3.5.7.c • Exemplo com xor (1).

Uma aplicação muito conhecida do **xor** é em criptografia. O **xor**, isoladamente, é usado no mais elementar dos algoritmos de **criptografia simétrica**. Também é usado, combinado com outros recursos, em outros algoritmos.

Sabendo-se que um **xor** entre dois **bits** retorna falso se eles são iguais e verdadeiro se eles são diferentes, podemos facilmente concluir que:

- se eu tenho um determinado texto e uma determina chave, e opero um **xor** bit a bit entre os dois, eu obtenho um resultado (criptografado) onde:
  - os **bits iguais** no texto e na chave resultaram em 0(**zero**)
  - e os **diferentes** resultaram em 1(**um**).
- se, agora, eu opero o **xor** entre o resultado criptografado e a chave, tenho exatamente o caminho inverso, **restaurando o texto original** (descriptografando-o), pela mesma razão que atuou no processo original de criptografia (inversão por contraste).
- **Obs.:** este é **apenas um exemplo de uso do xor**, pois para criptografia profissional, há algoritmos bem mais robustos.

*Vejamos o exemplo:*

```

#include <iostream>
#include <string>

class Cripta
{
public:
    enum { MAX_KEY = 32 };
private:
    char m_Key[ MAX_KEY + 1 ];
    size_t m_Size;
public:
    Cripta() // construtora
        : m_Size ( 0 )
    {
        *m_Key = 0;
    }

```

```

    m_Key[ MAX_KEY ] = 0;
}
void SetKey( const char * Key ) {
    strncpy( m_Key, Key, MAX_KEY );
}
const char * GetKey() const { return m_Key; }
void SetSize( size_t Size ) { m_Size = Size; }
size_t GetSize () const { return m_Size; }

// Função "Cript": para criptografar e descriptografar
// (pois a lógica é simplesmente simétrica):
void Cript( const char * Text, char * Output)
{
    const char * keyPtr ;
    size_t size = m_Size;

    for ( keyPtr = m_Key; size>0; ++Text, ++Output, --size )
    {
        // xor bit a bit entre byte entrada e correspondente na chave:
        *Output = *Text ^ *keyPtr; // ⊕
        // evoluir a posição da chave:
        ++keyPtr;
        if ( !*keyPtr ) // se atingiu o terminador zero...
            keyPtr=m_Key; // voltar ao primeiro byte da chave
    }
} // fim da função "Cript"

} // fim da declaração da classe "Cripta"

int main()
{
    using namespace std;

    char Text[255], Password[255], CriptText[255];
    size_t Size;

    cout << "informe texto e senha" << endl;
    cin.getline(Text, sizeof(Text));
    cin.getline(Password, sizeof(Password));
    cout << endl;
    Size= strlen(Text);

    Cripta crpt ; // cria um objeto "Cripta"
    crpt.SetKey ( Password ); // informa a senha
    crpt.SetSize( Size ); // informa o tamanho do texto a criptografar

    // 1) Criptografar de Text para CriptText:
    crpt.Cript( Text, CriptText);
    CriptText[Size] = 0x0;
    cout << "\nCriptografei (pode ter caracteres estranhos ou "
        "terminador zero no meio...):\n" << CriptText << endl;

    // 2) Des-criptografar de CriptText para Text:
    crpt.Cript(CriptText, Text );
    cout << "\nDescriptografei: \n" << Text << endl << endl;
    return 0;
}

```

Um outro exemplo do mesmo princípio (*simetria*) pode ser visto numa interessante implementação da função “swap” usando o operador **xor**.

Essa função clássica troca o valor de duas variáveis. A sua implementação mais comum é a que está na função **“swap\_1”** do exemplo abaixo. Em seguida temos uma função **“swap\_2”**, usando o **xor**.

```
#include <iostream>
template < typename T > void swap_1(T & a, T & b)
{
    T tmp = a;
    a = b;
    b = tmp;
    // e "b" ficara com o valor original de "a"
}
template < typename T > void swap_2(T & a, T & b)
{
    a ^= b ; // "a" contem a inversao de "a" e de "b"
             // e desse modo permitrá recuperar 'a' e 'b' trocados
    b ^= a;  // b = a
    a ^= b;  // a = b
}

int main()
{
    using namespace std;

    int a = 5, b = 6;
    cout << "- swap_1: valores originais: " << a << ", " << b << endl;
    swap_1( a, b );
    cout << "- swap_1: valores trocados : " << a << ", " << b << endl;

    cout << '\n' ;

    a = 5, b = 6;
    cout << "- swap_2: valores originais: " << a << ", " << b << endl;
    swap_2( a, b );
    cout << "- swap_2: valores trocados : " << a << ", " << b << endl;

    return 0;
}
/*
RESULTADO:
    - swap_1: valores originais: 5, 6
    - swap_1: valores trocados : 6, 5
    - swap_2: valores originais: 5, 6
    - swap_2: valores trocados : 6, 5
*/
```

#### 15.3.5.7.d • Exemplo com xor (2).

Um outro exemplo de uso prático da inversão verdadeiro/falso ocasionada pelo **xor**, é o de descobrir qual é o **último dia de cada mês**.

Vimos esse exemplo no exercício *“class data”*, mas a operação com o **xor** não foi ali explicada com detalhes.

É o que faremos aqui.

**Analisando o problema do “último dia de cada mês”, descobrimos três situações diferentes:**

1) Quando o mês é fevereiro teremos 29 dias se o ano for bissexto ou, do contrário, 28.

- Então poderíamos primeiro escrever uma função chamada **Bissexto**, que retornaria falso (**zero**) se o ano não for bissexto, e retornaria verdadeiro(**um**) se o ano for bissexto:
- Agora bastaria fazer:

```
if ( mes == 2 )
    return 28 + Bissexto ( Ano ); // Ou seja: 28 + um (ano bissexto),
                                // ou 28 + zero (não bissexto)
```

2) De janeiro a julho sabemos que os meses **pares** têm **30** dias e os **ímpares** têm **31**. Então é simples:

```
if ( mes <= 7 )
    return 30 + ( mes & 1 ); // O operador de bits and retornará um
                           // se o mes for ímpar.
    // Assim, teremos : meses ímpares = 30 + um ;
    // e meses pares = 30 + zero
```

3) De agosto a dezembro ocorre o **contrário**: meses **pares** têm **31** dias e os **ímpares** têm **30**:

```
if ( mes > 7 )
    return 30 + ! ( mes & 1 ); // O operador de bit's and retornará um
                              // se o mes for ímpar.
    // Em seguida usamos o operador lógico not ( ! ) para inverter
    // zero para um, ou um para zero (já que os pares tem 31 dias)
    // Assim, teremos : meses pares = 30 + um ;
    // e meses ímpares = 30 + zero
```

**E poderíamos também fazer (economizando uma operação):**

```
if ( mes > 7 )
    return 31 - ( mes & 1 ); // O operador de bitss and retornará um
                           // se o mes for ímpar.
    // Neste caso, não precisaríamos inverter o resultado do and.
    // Pois teremos : meses pares = 31 - zero ;
    // e meses ímpares = 31 - um
```

A pergunta é: existe algum meio de unir as duas última situações em uma **única operação sem usar o if**? Sim, através do **xor**. Para entender, vamos ver a seguinte tabela.

Meses	Cálculo necessário	Resultado final
6 (junho)	30 + zero	30
7 (julho)	30 + um	31
8 (agosto)	30 + um	31
9 (setembro)	30 + zero	30

*Veja agora como obter o resultado desejado através do xor:*

Meses	É ímpar ? (mes & 1)	É maior que 7(julho) ? ( mes > 7 )	Agora será efetuado um XOR entre as duas colunas ante- riores	Resultado final (30 + coluna anterior):
6(junho)	zero	zero	zero (estão iguais)	30 (30 + zero)
7(julho)	um	zero	um (diferentes)	31 (30 + um)
8(agosto)	zero	um	um (diferentes)	31 (30 + um)
9(setembro)	um	um	zero (iguais)	30 (30 + zero)

Vamos então implementar a solução.

```
// Primeiro precisamos de uma função para determinar se um ano é bissexto:
bool Bissexto(short nAno)
{
    // um número é bissexto se for divisível por 4 mas não por 100
    // (exceto se também for divisível por 400)
    // podemos usar o operador and bit a bit, para saber se é divisível por 4:
    return (( nAno & 3) == 0) &&
           ((nAno % 100) != 0 || (nAno % 400) == 0);
    // já para saber se é divisível por 100 ou por 400, não tem jeito,
    // temos que usar o resto de divisão.
}

// Agora, já podemos encontrar o último dia de um mês:
enum {FEVEREIRO = 2, JULHO = 7 };
char ultimoDiaMes (char Mes, short Ano )
{
    // se for fevereiro: 28 ou 29 dias; de janeiro a julho ( Mes <=7), os meses
    // ímpares tem 31 dias e os pares tem 30; agosto a dezembro (Mes>7),
    // os meses ímpares tem 30 dias e os pares tem 31
    return ( nMes==FEVEREIRO) ? 28+Bissexto(nAno)
                               : 30 + ( (nMes & 1) ^ (nMes > JULHO) );
    // usando o xor bit a bit
}

```

## 15.4 • Controles do fluxo de processamento.

### 15.4.1.1 • Parâmetros para funções e retorno de funções.

#### 15.4.1.2 • Declarando parâmetros.

Uma função pode ou não receber parâmetros, isto é, informações de que necessitará para realizar o seu trabalho. Um parâmetro de função também deve ser **declarado enquanto variável**, devendo a declaração indicar o seu **tipo**, como ocorre em qualquer declaração de variável.

#### 15.4.1.3 • Formas de declaração.

Em C há **duas** formas de declarar **variáveis** que **cumprem o papel de parâmetros** de função. Mas em C++ **há apenas uma**.

##### 15.4.1.3.a • Forma de declaração válida em em C e C++

```
int SomaNumeros ( int A , int B )    /* as variáveis A e B são parâmetros */
{
    int C;                          /* a variável C não é um parâmetro */
    INSTRUÇÕES ..... ;
}
```

##### 15.4.1.3.b • Forma de declaração inválida em C++

Existe uma outra forma de declaração, mas ela é válida apenas em C – e **não** é válida em C++.

```
int SomaNumeros ( A , B )
    int A ;                      /* a variável A é um parâmetro */
    int B ;                      /* a variável B é um parâmetro */
{
    int C;                      /* a variável C não é um parâmetro */
    INSTRUÇÕES ..... ;
}
```

O que há em **comum** entre essas duas formas de declaração das variáveis que servem como parâmetros é que, em ambos os casos, a declaração das variáveis-parâmetro é **feita antes da chave que abre** o bloco de instruções nomeado pela função.

A primeira delas sempre foi a mais usada por ser mais clara e simples. Felizmente, C++ **não manteve a compatibilidade com C** neste aspecto, e, devido à falta de clareza da segunda alternativa, ela **não** é válida nesta linguagem.

#### 15.4.1.4 • Funções com quantidade fixa de parâmetros.

O mais habitual (e melhor) é quando uma função tem uma quantidade fixa de parâmetros, todos com os seus tipos claramente estabelecidos.

Assim, a função *SomaNumeros*, do exemplo acima, deverá ser chamada sempre com os parâmetros adequados.

E isto significa que se uma função foi definida para receber dois parâmetros do tipo **int** ela só deve ser chamada mediante a passagem de dois valores desse tipo.

Assim, uma chamada à função `SomaNumeros` poderia ser escrita:

deste modo:

```
SomaNumeros ( 5 , 4 ) ;
```

ou deste:

```
{
    int A = 5 ; int B = 4 ;
    SomaNumeros ( A , B ) ;
}
```

mas **não** assim (provocará uma advertência do compilador):

```
{
    float A = 5.5 ; int B = 4 ;
    SomaNumeros ( A , B ) ;    /* AVISO! a variável "A" é do tipo float:
                                possível perda de precisão na
                                passagem para int */
}
```

Contudo, poderia assumir a perda de precisão explicitamente através da conversão (“**cast**”):

```
SomaNumeros ( (int)a , b ) ;
                OU
```

```
SomaNumeros ( int( a ) , b ) ; // apenas C++
```



A **chamada** a uma função com quantidade fixa de parâmetros deve obedecer tanto a essa quantidade quanto aos tipos dos parâmetros da função.

### 15.4.1.5 • Funções com quantidade variável de parâmetros.

Embora de um modo geral o uso desta espécie de função seja considerado um estilo pobre de programação, tanto **C** quanto **C++** admitem funções com uma quantidade variável de parâmetros.

Esse é o caso de funções como **printf** e **scanf**, por exemplo.

Nesse tipo de função devemos ter **pelo menos um** primeiro parâmetro **fixo** (que servirá para indicar quantos parâmetros teremos em seguida).

A partir daí é responsabilidade do programador extrair os demais parâmetros da pilha.



**Evite** criar funções com quantidade de parâmetros variável. Elas impedem que o compilador analise se estão sendo chamadas corretamente

#### 15.4.1.5.a • Declarando os parâmetros desconhecidos

Na declaração de parâmetros desta espécie de função devemos ter um ou mais parâmetros fixos, separados normalmente por vírgulas.



Após o último deles, deve existir uma vírgula seguida de reticências (que simbolizam uma lista com quantidade variável de parâmetros).

**Exemplo:**

```
double Media( int NumParams , ... )
```

#### 15.4.1.5.b • Exemplo de função com quantidade de parâmetros variável.

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

// O primeiro parâmetro, fixo, informa a quantidade de parâmetros adicionais.
// Em seguida, temos uma vírgula e reticências.
double Media( int NumParams , ... )
{
    // inicia a lista de parâmetros
    va_list Param;
    va_start( Param, NumParams );

    int iSoma = 0;

    for ( int iC=0; iC < NumParams; iC++)
    {
        iSoma += va_arg( Param, int);
        // Na linha acima, os parâmetros são, um a um,
        // extraídos da lista
    }

    va_end( Param ); // libera a lista de parâmetros
    return ( iSoma / NumParams );
}

// Usando a função com quantidade de parâmetros variável:

int main()
{
    // Abaixo, será chamada a função Media,
    // com quantidade de parâmetros variável.

    // Na chamada da função, o primeiro parâmetro indica a quantidade
    // de parâmetros adicionais. Neste exemplo temos quatro:

    cout << Media ( 4 ,
                    8 , 10 , 6 , 8 ) << endl;

    return 0;
}
```


#### 15.4.1.6 • Valor de retorno, tipo e protótipo de funções.

##### 15.4.1.6.a • Valor de retorno.

Há funções que apenas realizam uma tarefa.

Por exemplo: uma função denominada “**LimpaTela()**”, simplesmente imprime brancos na tela e retorna sem devolver qualquer valor. Sua tarefa não implica em um **resultado** formal.

Já uma função como a *SomaNumeros* tem como tarefa somar dois dados do tipo `int`. Assim, ela só faz sentido se informar para quem a chamou qual foi o resultado dessa operação de soma

 Isso significa que além de **receber** informações na forma de parâmetros, uma função também pode **retornar** uma (e **somente uma**) informação.

Desse modo a função *SomaNumeros* deveria ser escrita assim:

```
int SomaNumeros( int A , int B )
{
    int C = A + B
    return C;      /* retorna o resultado da soma */
                  /* logo, a função SomaNumeros retorna um valor do tipo int */
}
```

Deve ficar claro que a declaração de retorno (**return**) só pode ser associada a **um único valor**.

O valor de retorno define o tipo da própria função. Assim ela só pode retornar um único dado (o tipo desse dado determina o **tipo da função**).


Quando uma função retorna um valor ocorrerá que uma **cópia** desse valor será armazenada em uma **memória temporária de retorno**. Nesse momento, essa memória poderá ser utilizada em qualquer operação pendente (por exemplo, uma atribuição). Por isso fazem sentido as seguintes expressões:

`C = SomaNumeros ( A , B ) ;` /\* o valor de retorno da função é copiado para a variável C \*/

*OU*

`C = SomaNumeros ( A , B ) * 5 ;` /\* o valor de retorno da função é primeiro multiplicado por 5 e o resultado final será então atribuído à variável C \*/

#### 15.4.1.6.b • Tipo de uma função.

 Utilizando o seu **retorno formal** (ou **resultado**), uma função pode retornar um **único** valor, o qual determina o seu tipo.  
Ou seja: **o tipo de uma função** é determinado pelo **tipo do valor que ela retorna**.

E se ela não retornar um valor (como no exemplo acima, da função *LimpaTela*), dizemos que essa função é do **tipo void** (ausência de valor de retorno). Desse modo, dizemos que a seguinte função é do tipo `int`:

```
int SomaNumeros( int A , int B )
{
    int C = A + B ;
    return C ;      /* retorna "C" que é um valor do tipo int */
}
```

- Desse modo quando chamamos uma função devemos observar:

- que os parâmetros passados o sejam na mesma quantidade e com os mesmos tipos conforme serão recebidos.
- mas **também** que o **retorno** seja recebido de modo compatível; não é possível armazenar (sem perda de precisão) o retorno de uma função do tipo float em uma variável do tipo int.

### 15.4.1.6.c • Protótipo de funções.

Quando queremos usar uma função como **printf**, por exemplo, precisamos informar ao compilador qual é o modo correto de chamada a essa função. Isto para que ele possa analisar se todas as chamadas feitas à função estão corretas.

Por isso, sempre que usamos **printf**, precisamos incluir o arquivo **stdio.h**. Porque neste arquivo está **descrito** o modo correto de emprego da função.

E essa **descrição** é estabelecida através do **protótipo** da função.

- Um protótipo é uma máscara que define exatamente a configuração de uma determinada função;
- Isso significa que um protótipo deve especificar os seguintes aspectos da função:
  - ❑ quantidade de parâmetros;
  - ❑ tipo de cada parâmetro;
  - ❑ tipo do retorno (tipo da função).
- Desse modo o protótipo:
  - **documenta** como deve ser usada a função;
  - permite que o **compilador cheque** se todas as chamadas a uma determinada função estão corretas).

#### **Exemplo:**

O protótipo da função *SomaNumeros* deveria ser escrito assim:

```
int SomaNumeros( int , int ) ;
```

- a função recebe dois parâmetros;

- os dois são do tipo int;


- e o tipo da função é int (logo ela retorna um valor int).

### 15.4.1.7 • Modos de passagem de parâmetros (recapitulando)

⇔ Há **três** modos de passar parâmetros para uma função:

- ❑ 1 - por valor;
- ❑ 2 - por endereço;
- ❑ 3 - por referência.

#### 15.4.1.7.a • 1 - Passando parâmetros por valor.

 A função que recebe parâmetros recebe apenas **cópias de valores, nada sabendo sobre os endereços** em que tais valores estão armazenados no contexto de quem chamou a função.

*chamada da função (envia dados):*

```
int main( )
{
    int A = 5 ; int B = 4 ;
    SomaNumeros ( A , B ) ;
    return 0;
}
```

*definição da função (recebe os dados como parâmetros)*

```
int SomaNumeros ( int A, int B )
{
    return A + B ;
}
```

- as variáveis **A** e **B** na função main( ) são **locais** e **privativas** do seu contexto;
- **o mesmo ocorre** com **A** e **B** em SomaNumeros;
- ou seja: embora, por coincidência, tenham o mesmo nome, são regiões de memória diferentes.
- o que ocorre então é **uma cópia o conteúdo** da primeira variável da chamada para a primeira variável da recepção - e igualmente para a segunda.
- Por isso, dizemos que os parâmetros aqui foram passados por **valor**.

❑ *Veja o seguinte mapa:*

<i>Endereço de memória (hipótese)</i>	<i>Variável</i>	<i>Valor</i>
2000	<b>A</b> (main)	<b>5</b>
2004	<b>B</b> (main)	<b>4</b>
5000	<b>A</b> (somaNumeros)	<b>5</b> (recebeu cópia de conteúdo)
5004	<b>B</b> (somaNumeros)	<b>4</b> (recebeu cópia de conteúdo)

		do)
--	--	-----

- ❑ Os endereços são diferentes. Logo são variáveis independentes.
- ❑ Qualquer alteração que SomaNumeros( ) faça nas suas variáveis-parâmetro **em nada afetar**á as variáveis de main( ).

#### 15.4.1.7.b • 2 - Passando parâmetros por endereço:

✎ Neste caso, as variáveis-parâmetro, ao invés de receber uma cópia dos valores passados por quem chamou a função, recebem uma cópia dos endereços onde esses valores estão armazenados.

- Como já vimos, o que torna possível trabalhar diretamente com o endereço de uma variável, em C/C++, são os operadores de **endereço (&)** e **ponteiro (\*)**.

*Assim se escrevermos:*

```
1 int B ;
2 int * A ;
3 A = &B ;
```

*estaremos ordenando:*

- 1 Reserve memória para um valor do tipo int (e a apelide de “B”).
- 2 Reserve a memória necessária para armazenar um endereço (e a apelide de “A”);  
e garanta que nessa memória sejam gravados endereços para valores do tipo int.
- 3 Armazene em “A” o endereço de “B” [ A = &B ].

✎ O que é válido para uma variável qualquer é válido também para uma variável colocada no papel de parâmetro de função. Assim só podemos passar endereços para as funções que estejam aptas a receber endereços. Por isso:  
*As variáveis-parâmetro que pretendem receber endereços precisam ser **ponteiros**.*

- Então, uma função que recebe endereços deve estar:

- 1) definida de modo apropriado e
- 2) deve ser chamada de modo apropriado.

**chamada da função (envia endereços):**

```
int main( )
{
    int A = 5 ; int B = 4 ;
    SomaNumeros ( &A , &B ) ;
    return 0;
}
```


**definição da função (recebe endereços em variáveis-ponteiro)**

```
int SomaNumeros ( int * A , int * B )
{
    int C = * A + * B ;
```


```
    return C;
}
```

Observe que a linha de instrução escrita acima [  $C = *A + *B$  ] significa:

- 1) vá ao endereço guardado em “A” e pegue o valor ali armazenado;
- 2) vá ao endereço guardado em “B” e pegue o valor ali armazenado;
- 3) some os dois valores e armazene o resultado em “C”.

 Se o operador ponteiro (\*) não estivesse presente estaríamos ordenando que o conteúdo da variável “A” (um endereço) fosse somado ao conteúdo da variável “B” (outro endereço). Isso não faria o menor sentido: estaríamos pretendendo que a variável “C” armazenasse a soma de dois endereços!!!

Mas, felizmente, nem chegaria a ocorrer, pois o compilador acusaria o erro: manipulação incorreta de tipos de dados.

 Cabe aqui uma observação importante:  
Se estamos passando um endereço como parâmetro isto significa que a função que recebe esse parâmetro passa a ter acesso a uma posição de memória pertencente ao contexto de quem chamou a função.

Logo, esta função poderá alterar o conteúdo dessa posição de memória pois, neste caso, ela está sendo **enxergada**.

- na passagem de parâmetros por endereço o que ocorre então é **uma cópia do endereço da variável e não o seu conteúdo** (o valor).
- as variáveis de *main* e as variáveis-parâmetro de *SomaNumeros* são variáveis independentes (ocupam posições de memória diferentes); mas como as variáveis de *SomaNumeros* estão de posse dos **endereços** de variáveis pertencentes a *main*, *SomaNumeros* dispõe agora de um **meio de acesso** às variáveis de *main*, podendo então alterá-las.

□ *Veja o seguinte mapa:*

Considerada esta situação:

```
int main( )
{
    int A = 5 ; int B = 4 ;
    SomaNumeros ( &A , &B ) ;
    return 0;
}
int SomaNumeros ( int * A , int * B )
{
    .....
}
```

Poderíamos ter a seguinte disposição de memória:

Endereço de memória (hipótese)	Variável	Conteúdo
2000	A (main)	5 (um valor int)
2004	B (main)	4 (um valor int)

5000	*A (somaNume- ros)	<b>2000</b> (recebeu cópia do endere- ço)
5004	*B (somaNume- ros)	<b>2004</b> (recebeu cópia do endere- ço)

- ❑ Os endereços das variáveis são diferentes. Logo são variáveis independentes.
  - ❑ Contudo, as variáveis de SomaNumeros( ) armazenam os endereços das variáveis de main.
  - ❑ Desse modo SomaNumeros( ) poderá alterar o conteúdo das variáveis em main( ).
-

**15.4.1.7.c • 3 - Passando parâmetros por referência:**

✎ Embora este modo de passagem de parâmetros aparente alguma semelhança com a passagem por endereço, na verdade são dois modos completamente diferentes.

✎ Na passagem por referência não temos uma segunda posição de memória armazenando um endereço e sim um **segundo nome para uma mesma posição de memória**.

□ Exemplo da diferença entre passagem por endereço e referência:

*por endereço:*

```
int main( )
{
    int A;
    Segunda ( &A ) /* captura o endereço de "A" e o envia como parâmetro para a função segunda */
    return 0;
}
void Segunda( int * X ) /* a função segunda está apta a receber o endereço de um valor do tipo int */
```

□ desse modo, (hipótese) se “A” fosse o nome do endereço 2000 na memória, “X”, (que, por sua vez, seria o nome do endereço 2004), armazenaria 2000.

A (nome de 2000)	X (nome de 2004)
armazena o valor 5	armazena o endereço 2000

*por referência:*

□ neste caso, a função que receberá parâmetros deverá declará-los como o **operador de referência (&)**

```
int main( )
{
    int A;
    Segunda ( A )
    return 0;
}
void Segunda ( int & X ) /* devido ao operador de referência, o compilador considera "X" como um segundo nome para a posição de memória que, em main, recebeu o nome "A" */
```

□ desse modo, (hipótese) se “A” fosse o nome do endereço 2000 da memória, “X” seria um **segundo nome para 2000**.

A (nome de 2000)	X (segundo nome para 2000)
Armazena o valor 5	



### 15.4.1.8 ▪ Há dois motivos para passar parâmetros por endereço ou por referência.

#### 1) Retornos extras

Como uma função tem apenas um retorno formal podemos precisar de “**retornos-extras**”.

- ❑ Não há como fazer isso contando com o retorno formal de função (pois este permite um e apenas um retorno).
- ❑ Mas, se passarmos o endereço de variáveis, a função chamada poderá acessar e alterar essas variáveis produzindo efeitos no contexto de quem chamou.
- ❑ É assim que atua a função `scanf( )`:
  - Ela já tem um retorno, que informa se a operação de entrada de dados foi bem sucedida.
  - Mas precisamos de um “retorno-extra”, sem o qual a função não faria sentido: o resultado da própria entrada de dados, que é sua tarefa real.
  - por isso o segundo parâmetro de `scanf( )` deve ser passado com o operador de endereço (`&`).
  - desse modo essa variável será “preenchida” por `scanf( )`.

#### 2) Performance

O segundo motivo é **performance** (velocidade de cópia na passagem de parâmetros).

- ❑ Se estamos passando um `int`, um `char`, ou mesmo um `float`, é preferível passar por valor. Pois estamos lidando com um número reduzido de bytes.
- ❑ Mas podemos criar nossos próprios tipos de dados. São as estruturas de dados que funcionam como registros com diversos campos, podendo conter vários **int**, ou vários **char**, etc.
- ❑ Dados desse tipo podem ocupar grande quantidade de memória e nesse caso a cópia por valor (o que significa a cópia de todos os campos), será bem mais lenta.
- ❑ Além disso uma cópia por valor duplica -ainda que momentaneamente- o uso da memória (é preciso manter o “original” e a “cópia” simultaneamente. Isto porque quem chamou a função usou uma memória (e ela ainda está viva); e a própria função reservou uma memória para receber uma cópia desse valor (o parâmetro) que, naturalmente, também está vivo nesse momento.
- ❑ Nesse caso se passarmos uma cópia de endereço estaremos passando apenas **dois ou quatro bytes**, que é o espaço necessário para armazenar um endereço (dois ou quatro bytes para sistemas de 16 bits; e sempre quatro bytes para sistemas de 32 bits).

### 15.4.1.9 ▪ Impedindo efeitos colaterais desnecessários.

#### □ ATENÇÃO:

➤ Quando utilizarmos passagem por endereço ou por referência apenas pelo motivo “performance” (não havendo nenhum interesse em “retornos extras”) abrimos uma porta para efeitos colaterais.

□ Não queremos que a função chamada possa alterar memórias de outros contextos.

Queremos apenas velocidade.

□ E como garantir que a função chamada **seja impedida** de promover **alterações não previstas ou não desejadas** em outros contextos?

➤ Através do especificador **const**.



Uma função que recebe endereços ou referências apenas pelo motivo “performance”, deve ter direito de acesso ao endereço recebido **somente para leitura** e não para gravação.



O especificador **const** determina essa restrição, estabelecendo um status “constante” (*read-only*) para a variável.

➤ Desse modo a função SomaNumeros(usando ponteiros como parâmetros) poderia ser escrita assim:

```
int SomaNumeros ( const int * A , const int * B )
{
    int C = * A + * B ;
    return C;
}
```

➤ E, se usasse referências como parâmetros, poderia ser escrita assim:

```
int SomaNumeros ( const int & A , const int & B )
{
    int C = A + B ;
    return C;
}
```

### 15.4.2 ▪ Lógica estruturada através de objetos.

Ponteiros para função, conforme vimos acima, são um recurso importante.

Mas em C++ raramente iremos utilizar esse recurso **diretamente**.

Isto porque a linguagem suporta a criação automática de ponteiros para função mediante um recurso denominado “**funções virtuais**”.

E a utilização destes ponteiros implementados pelo próprio compilador ocorrerá de forma indireta, mais disciplinada, através da lógica interna das **estruturas C++**.

Mas isso não significa que perdemos tempo no item anterior aprendendo como funcionam os ponteiros para função. Pois, sem essa compreensão, seria praticamente impossível entender, mais tarde, como **realmente** trabalham as “funções virtuais”.

Este assunto será tratado na próxima seção (orientação a objetos em C++).

Mas, por enquanto, vamos deixar **registrado** que as **estruturas C++ afetam o fluxo de processamento** através de uma lógica própria:

- a. Disparo automático de funções construtoras sempre que uma variável estruturada é **criada**.  
Já vimos funções construtoras na primeira parte desta apostila; mas precisaremos voltar ao assunto.
- b. Disparo automático de funções destrutoras sempre que uma variável estruturada é **liberada**.
- c. A tabela de ponteiros para funções ditas virtuais.

Na próxima seção veremos os itens acima em detalhe.

E aproveitaremos então para reescrever o exercício “**Relatório Padrão**” utilizando os melhores recursos da estrutura C++ e acrescentando novos atributos e funções.

### 15.4.3 • Controles de laço (recapitulando).

Um **laço** permite **repetir** um **bloco** de linhas de instrução enquanto uma **condição** for verdadeira.

Já utilizamos os laços “**for**” e “**while**” em capítulos anteriores. Vamos agora precisar melhor a sintaxe desses laços e ver o que ainda falta.

#### 15.4.3.1 • O laço *for*.

##### 15.4.3.1.a • Forma geral e funcionamento.

for ( <Início> ; <condição para continuidade> ; <progressão> )

Onde:

- **<Início>** : é executado **apenas na primeira vez** (antes do laço iniciar).
- **<Condição para continuidade>** : é executada **antes de cada passada do laço**, logo é executada **também na primeira vez** (antes do laço iniciar)
- se a avaliação resultar verdadeira, o bloco de instruções associado ao laço será executado; do contrário ocorrerá um salto para a primeira instrução após o fim do laço.
- **<Progressão>** é executada **ao final** de cada passada do laço (logo **não é** executada na primeira vez).

*Exemplo:*

```
#include <iostream>
int main()
{
    for ( int Contador = 1 ; Contador <= 3 ; Contador++)
        cout << "Valor de Contador = " << Contador << endl ;

    cout << "Valor de Contador apos o fim do laço = "
         << Contador << endl ;

    return 0 ;
}
/*
    RESULTADO
    Valor de Contador = 1
```

Valor de Contador = 2  
 Valor de Contador = 3  
 Valor de Contador apos o fim do laço = 4

\*/

*Analisando o resultado:*

- **<Início>** é executado antes de qualquer coisa; e não mais será executado.
- **<Condição para continuidade>** é executada em seguida. O valor de **Contador** neste momento é **1**. Como está **menor que 3**, a avaliação retorna **verdadeiro** e o laço é iniciado (do contrário ocorreria um salto para o fim do laço e ele **não seria** executado **se-quer uma vez**).
- O bloco de instruções associado ao laço é executado e a linha "*Valor de Contador = 1*" é impressa.
- **<Progressão>** é executada. **Contador** é incrementado para **2**.
- **<Condição para continuidade>** é executada. O valor de **Contador** neste momento é **2**. Como está **menor que 3**, a avaliação retorna **verdadeiro** e o laço continua.
- A linha "*Valor de Contador = 2*" é impressa.
- **<Progressão>** é executada. **Contador** é incrementado para **3**.
- **<Condição para continuidade>** é executada. O valor de **Contador** neste momento é **3**. Como está **igual a 3**, a avaliação retorna **verdadeiro** e o laço continua.
- A linha "*Valor de Contador = 3*" é impressa.
- **<Progressão>** é executada. **Contador** é incrementado para **4**.
- **<Condição para continuidade>** é executada. O valor de **Contador** neste momento é **4**. Como está **maior que 3**, a avaliação retorna **falso** e o laço é interrompido, com um salto para a primeira linha após as chaves de fechamento do bloco de instruções associado.
- Imediatamente pós o fim do laço, **cout** imprime o valor de **Contador**. Pelo que vimos acima, o valor impresso neste ponto só poderia ser **4**.

*Em Resumo:*

- **<Início>** [ **int Contador = 1** ] – foi executado **uma vez** (antes do laço iniciar);
- **<Condição para continuidade>** foi executada **antes** de cada passada no laço; portanto, foi executada uma vez a mais (momento em que a condição ficou falsa) que o número de execuções do laço. Logo foi executada **quatro vezes**.
- **<Progressão>** foi executada **ao final** de cada execução das instruções associadas ao laço. Logo foi executada **três vezes**.

### 15.4.3.1.b • Sintaxe.

#### a. A estrutura de controle:

for ( [ ... , ] [ ... ] ; [ ... , ] [ ... ] ; [ ... , ] [ ... ] )				
<Início>	;	<condição continuidade>	;	<progressão> ou <passo>

- A estrutura de controle do **laço for** é composta por **três segmentos** ( <início>, <condição para continuidade> e <progressão> ), separados por **ponto e vírgula**.
- Cada um dos três segmentos **pode** conter **uma, nenhuma** ou **diversas instruções**, separadas por **vírgula**.

*Exemplo 1 ( o TERCEIRO segmento contem DUAS instruções):*

```
int StrCopia ( char * sDestino, const char * sOrigem, int nMaxBytes )
{
    const char * sOrigemInicial ;
    for ( sOrigemInicial = sOrigem ;
          *sOrigem != '\0' && sOrigem - sOrigemInicial < nMaxBytes ;
          ➡ ++sOrigem , ++sDestino )
```

```

{
    *sDestino = *sOrigem ;
}
*sDestino = '\0' ;
return sOrigem - sOrigemInicial; // retorna:
                                // quantidade de bytes copiados.
}

```

Exemplo 2 ( o PRIMEIRO segmento não contém NENHUMA instrução):

```

double Fatorial_UsandoFor( int Numero )
{
    double Fatorial = 1;

    ↓

    for (      ; Numero > 1      ; --Numero )
        Fatorial *= Numero;

    return Fatorial;
}

```

No exemplo acima, como a variável utilizada para o controle de laço já está com o valor correto para iniciar o laço (pois é o próprio valor que foi passado como parâmetro), não é necessário usar o segmento <Início> para essa variável (embora possa ser usado para outras, se for o caso).

#### b. Instruções associadas:

- **Diversas instruções associadas:** se precisamos executar diversas instruções, elas devem compor um bloco e este bloco deve estar imediatamente em seguida à estrutura de controle do **laço for**; desse modo, o bloco estará automaticamente associado ao laço:

```

for ( ... ; ... ; ... )
{
    instrucao_1;
    instrucao_2;
    .....
}

```

- **Uma instrução única associada:** se precisamos executar apenas uma, não é necessário um bloco. A próxima instrução imediatamente seguinte à estrutura de controle do laço é automaticamente associada a ele:

```

for ( ... ; ... ; ... )
    instrucao_Unica;

```

- **Nenhuma instrução associada:** se **tudo** o que precisamos executar é a **própria estrutura de controle do laço** (não havendo assim **nenhuma** instrução associada), assinalamos isso incluindo um **ponto e vírgula** imediatamente em seguida à estrutura de controle (que com isso passa a ser uma instrução **completa**):

```

for ( ... ; ... ; ... ) ;
// devido ao ponto e vírgula final, nenhuma
// instrução abaixo será associada a este laço for

```

Exemplo de um laço for fechado pelo ponto e vírgula:

```

for ( int Conta = 1 ; Conta <= 12 && Pausa() ; ++Conta ) ;

↓
bool Pausa( )

```

```

{
    PararProcessamentoPorMilisegundos ( 1 ) ; // pausa
                                                // de 1 milisegundo

    if ( AlguemApertouUmaTecla ( ) )
        return false;
    else
        return true;
}

```

No exemplo acima, o laço `for` serve para esperar que alguma coisa ocorra no teclado durante um certo tempo. Se “*Conta*” atingir 12, o laço será interrompido; e ele será interrompido também se a função *Pausa* retornar falso.

A cada chamada à função *Pausa*, o processamento é interrompido por um milisegundo. Se nesse período algo ocorreu no teclado, a função retornará falso, interrompendo o **for**.

O número 12 funciona aqui como uma espécie de *timeout* (se nada ocorrer em um pouco mais que 12 milisegundos, então o laço deve desistir de esperar por eventos no teclado).



Atenção: no exemplo acima, o ponto e vírgula fechando o **laço for** foi colocado **intencionalmente**. Mas saiba que um erro comum cometido por programadores iniciantes na linguagem é colocar um ponto e vírgula após um **for** por **distração**, ocasionando resultados não desejados.

#### 15.4.3.1.c • Cuidados a tomar com o laço `for`.

- **Ponto e vírgula** fechando o laço quando isso **não** se aplica (ver comentário acima).
- Uso de **mais do que uma condição**, separadas por vírgulas. Observe os resultados dos dois exemplos abaixo.

`int A , B ;`

Exemplo 1:

```

for ( A= 1 , B=4 ; A < 3 , A < B ; A++ )
    cout << A << endl;

```

```

/* RESULTADO (executou TRÊS vezes)
    1
    2
    3
*/

```

Exemplo 2:

```

for ( A= 1 , B=4 ; A < B , A < 3 ; A++ )
    cout << A << endl;

```

```

/* RESULTADO (executou DUAS vezes)
    1
    2
*/

```



Quando colocamos diversas condições separadas por vírgulas em um laço **for**, todas são **executadas**, mas apenas a **última** é levada em conta como **resultado** da avaliação para determinar se o laço deve ter continuidade.

Caso quiséssemos **combinar** as duas comparações teríamos que utilizar o operador lógico adequado em uma **única** condição, por exemplo:

```

; A < B && A < 3 ; // AND
ou então:
; A < B || A < 3 ; // OR

```

Justamente porque existem várias formas de combinar comparações, não seria possível esperar que o compilador pudesse unir comparações soltas, separadas por vírgula.

### 15.4.3.2 • O laço “while”.

#### 15.4.3.2.a • Forma geral e funcionamento.

```
while ( <condição para continuidade> )
```

A estrutura de controle de um laço **while** é mais simples do que a do laço **for**. Temos apenas uma condição para continuidade.

Do **mesmo modo** que no laço **for**, ela é executada **antes** da execução das instruções associadas. E, do mesmo modo, se na primeira avaliação for obtido um resultado falso, o laço não será executado sequer uma vez.

Exemplo 1:

```

.....
int x = 1 ;
while ( x < 5 )
{
    cout << "x= " << x << endl;
    x++ ;
}
/* RESULTADO:
                                x= 1
                                x= 2
                                x= 3
                                x= 4
*/
*/

```

Exemplo 2:

```

.....
int x = 6 ;
while ( x < 5 ) // condição estará falsa na primeira avaliação
{
    cout << "x= " << x << endl;
    x++ ;
}
/* RESULTADO (nada foi impresso)
*/

```

#### 15.4.3.2.b • Sintaxe.

##### a. A estrutura de controle:

```
while ( [ ... , ] [ ... ] )
```

<condição continuidade>
----------------------------

No único segmento de controle (<condição para continuidade>) do laço **while**, podemos ter diversas instruções separadas por **vírgula**.

**b. Instruções associadas.**

As três situações abaixo são as mesmas de um **laço for**. Assim, tudo o que foi dito acima para o **for**, aplica-se aqui.

- **Diversas instruções associadas:** se precisamos executar diversas instruções, elas devem compor um **bloco**;

```
while ( condicao )
{
    instrucao_1 ;
    instrucao_2 ;
    .....
}
```

- **Uma única instrução associada:** se precisamos executar apenas uma, não é necessário um bloco. Será considerada a próxima instrução.

```
while ( condicao )
    intrucao_unica ;
```

- **Nenhuma instrução associada:** um ponto e vírgula imediatamente após a **estrutura de controle do laço** torna-a uma instrução **completa**; nenhuma outra instrução será associada.

```
while ( condicao ) ;
```

**15.4.3.2.c • Cuidados a tomar com o laço while.**

- **Ponto e vírgula** fechando o laço quando isso **não** se aplica (ver comentário feito para o laço **for**).
- Uso de **mais do que uma condição**, separadas por vírgulas. Do **mesmo modo que no laço for** apenas a segunda condição é considerada como resultado para a avaliação de continuidade. Observe os resultados dos exemplos abaixo.

Exemplo 1:

```
int a = 1 , b=4 ;
while ( a < 3 , a < b )
{
    cout << a << endl ; a++ ;
}

/* RESULTADO:
1
2
3
*/
```

Exemplo 2:

```
int a=1 , b=4 ;
while ( a < b , a < 3 )
    cout << a << endl ; a++ ;

/* RESULTADO:
1
2
*/
```



Em ambos os casos: era isso o que você queria?

Se realmente isso foi intencional, o seu código irá ficar extremamente **confuso** (o que é o primeiro indício de **código mal escrito**).



- Cuidado também ao aproveitar vírgulas na condição de continuidade para **alterar variáveis**:

```
int a=1 ;
while ( a++ , a < 4 )
    cout << a << endl;
    /* RESULTADO:
                                2
                                3
    */
```

Se você desejava imprimir **1, 2 e 3 não** deveria ter usado esse método. Pois a **primeira** instrução dentro da estrutura de controle ( **a++** ) é executada em primeiro lugar como uma **instrução independente**. Assim, mesmo o operador de incremento estando pós-fixado, a variável **a** é incrementada **antes** da comparação.

E, se você tentasse resolver isso, colocando a instrução de incremento ( **a++** ) em segundo lugar só iria piorar as coisas:

```
int a = 1;
while ( a < 4 , a++ )
    cout << a << endl ;
    /* RESULTADO:
        LOOP PRATICAMENTE INFINITO
    */
```

O **while** entende a **última instrução** da estrutura de controle como aquela cuja avaliação deve determinar a continuidade do laço.

E a última instrução no exemplo acima é **a++**.

Essa instrução se desdobra em duas:

- para o **while** ela deve dar um **resultado lógico (verdadeiro ou falso)**;
- e, além disso, ela deve ser incrementada (**++**).
- Ocorre então o seguinte:
- Como o operador de incremento está pós-fixado, o incremento será feito por último.
- Então, na primeira vez, o valor de "**a**" é **1** – e para a linguagem isso significa **verdadeiro** (diferente de zero).
- Agora **a é incrementado**, ficando com **2**.
- O laço é executado.
- Na próxima avaliação, sabendo-se que o valor de **a** é **2**, novamente o resultado da avaliação é **verdadeiro** (diferente de zero).
- Novamente "**a**" será incrementada e o laço será executado.
- E assim sucessivamente. Para fins práticos, temos um **loop** infinito.
- Ele só será interrompido porque, em algum momento, será estourada a capacidade de armazenamento do **int** com sinal (0x77777777 em 32 bits). Então a variável "**a**" assumirá um valor negativo e continuará sendo incrementado até atingir **zero**. E só nesse momento o laço será interrompido.
- E, se fizéssemos:

```
int a = 0 ; // inicializado com zero
while ( a < 4 , a++ )
    cout << a << endl ;
    /* RESULTADO:
        O LAÇO NÃO FOI EXECUTADO NENHUMA VEZ
    */
```

- Pois o **while** avaliou o valor de **a** como sendo a condição lógica de continuidade; na primeira vez **a** está com zero. Logo o resultado é falso.
- Em seguida a operação de incremento pós-fixada é executada e **a** fica com **1**. Mas o laço já havia sido encerrado.

### 15.4.3.3 ▪ Considerações gerais sobre os laços *for* e *while* (*tenha cuidado com...*)

- No caso do laço **for**, algumas vezes é bem conveniente o uso das vírgulas no primeiro segmento (**início**) ou no terceiro(**progressão**).
  - Contudo, exageros devem ser evitados: muitas vírgulas seguidas, em qualquer um desses dois segmentos tornam o código confuso e, em consequência, mal escrito.
- Quanto à condição de continuidade(segundo segmento no **for**; e segmento único no **while**):
  - tanto no caso do laço **for**, como no **while**, considere como “recurso de baixa legibilidade” o uso da condição de continuidade para incluir diversas instruções separadas por vírgulas.



Tanto no laço **for** como no **while**, **raramente (ou mesmo nunca)** utilize diversas instruções separadas por vírgulas dentro da condição de continuidade.



Um outro problema potencial diz respeito ao uso do **operador de atribuição dentro da condição de continuidade** dos laços. Analisaremos este problema **mais a frente**, ao falar do controle de decisão “**if**”, já que isto também se aplica a ele.

### 15.4.3.4 ▪ O laço “do ... while”.

#### 15.4.3.4.a ▪ Forma geral, funcionamento e sintaxe:

do <instruções associadas> while ( <condição de continuidade> ) ;

Detalhes da sintaxe:



Também é possível separar instruções com vírgulas na condição de continuidade. E com os mesmos problemas já levantados acima..

#### Funcionamento:



A principal diferença deste controle de laço para os anteriores é que a condição de continuidade é avaliada **no final**, após a execução das instruções associadas. Logo, o bloco será executado **pelo menos uma vez**.

#### Instruções associadas:

- **Diversas** instruções associadas:

```
int x=1;
do
```

```

{      // bloco de instruções
    cout << x << endl;
    x++ ;
}
while ( x < 4 ) ;

/* RESULTADO:

1
2
3

*/

```

- **Uma** única instrução associada:

```

int x=1;
do
    cout << x++ << endl; // Bloco não é obrigatório; portanto, sem chave.
                        // Mas aqui as chaves melhorariam a legibilidade.
while ( x < 4 ) ;
/* RESULTADO:

1
2
3

*/

```

- **Nenhuma** instrução associada:

- Neste caso isso **não faz sentido** pelo seguinte:

```

int x=1;
do ; // o ponto e vírgula aqui indica "nenhuma instrução associada"
    // e, neste caso, o "do" simplesmente está encerrado.
    // LOGO, essa linha é ignorada.

```

**E, se, em seguida, escrevêssemos:**

```
while ( x++ < 4 && Pausa( ) ) ;
```



Esta linha será tratada como um while (independente do "do" acima)

Portanto não faria sentido escrever:

```

do ; while ( x++ < 4 && Pausa( ) ) ;
//escrita correta, mas inútil. Bastaria:
while( x++ < 4 && Pausa( ) ) ;

```

- **E também não podemos escrever o "do" seguido diretamente pelo "while":**

```
do while ( x++ < 4 && Pausa( ) ) ; // escrita incorreta.
```

### ***Exercício: usando o laço "do ... while".***

Um bom exemplo seria uma variação do exercício "lista de números pares".

Se garantirmos, no início, que o primeiro e o último números a imprimir são pares e o primeiro não é maior que o último, então isso significa que **todos** serão impressos.

Portanto não é necessário fazer a avaliação no início do laço. Por isso pode ser usado o "do ... while" (a avaliação será feita no final).

***Para este exercício, siga os seguintes passos:***

- 1) Peça ao usuário que informe o primeiro e o último números a imprimir.

- 2) Garanta que eles sejam pares e o primeiro não seja maior que o último.
- 3) Usando “do ... while”, evolua do primeiro ao último imprimindo todos.

*Após concluir, compare com a solução abaixo.*

```
#include <iostream>
int main()
{
    int iPrimeiro , iUltimo;
    bool bDadosCorretos = false ;
    cout << "Imprimir lista de numeros pares" << endl;
    cout << "Informe o primeiro e o ultimo numeros a imprimir" << endl;

    // 1) No laço while abaixo serão alimentados o número inicial e o final
    while ( ! bDadosCorretos )
    {
        cin >> iPrimeiro >> iUltimo ;
        // Se um dos dois números não é par...
        if ( iPrimeiro & 1 || iUltimo & 1 )
            cout << "Digite apenas numeros pares\n" ;
        else if ( iPrimeiro > iUltimo ) // primeiro maior que último...
            cout << "Primeiro numero nao pode ser maior que o ultimo\n"
        else
            bDadosCorretos = true; // Isto fará com que
                                   // o while seja interrompido
    } // Fim do while para pegar dados.

    cout << '\n' << "Lista dos numeros pares entre " << iPrimeiro
        << " e " << iUltimo << endl;

    // 2) No laço "do ... while" abaixo, serão impressos os números pares.
    do // será executado pelo menos uma vez
    {
        cout << iPrimeiro << endl ; // Imprime.
        iPrimeiro += 2 ; // Evolui para o próximo número par.
    } while ( iPrimeiro <= iUltimo ) ; // Encerra aqui se a condição for falsa.
    return 0;
}
```



O criador de C++, Bjarne Stroustrup, considera o “do ... while”, uma fonte de erros e confusões. Isto porque, como o laço é sempre executado pelo menos uma vez, determinadas mudanças feita no código anterior ao laço podem modificar a situação, de tal modo que a primeira execução torne-se agora incorreta.

É o que aconteceria, no exercício acima, se alguém modificasse a primeira parte, suprimindo a análise que é feita para impedir que o número inicial seja maior que o final. Neste caso o primeiro número **passaria a ser impresso**, pelo fato de que a condição (**iPrimeiro <= iUltimo**) só é avaliada ao final.



Isto poderia ocorrer **principalmente** em situações semelhantes a essa, mas onde as duas partes do código estivessem escritas em **funções separadas** (e talvez até em **módulos separados**), de tal modo que, ao alterar a primeira parte, o programador **não visse** (ou não lembrasse) que, na segunda parte, é usado um “do ... while” ao invés de um “while”.  
Até porque o programador responsável pela alteração pode não ser o mesmo que criou o código original.

Mas sempre que não exista essa possibilidade de que o laço possa ser afetado por outro trecho de código, e nas situações em que sempre será preciso executar o laço pelo menos uma vez, então o “**do ... while**” é uma alternativa melhor que o “**while**”.

## 15.5 • Ponteiros, Matrizes e Referências

### 15.5.1.1 • Alocando livremente memória no *heap* (evite...)

Podemos solicitar que uma região de memória (localizada no *heap*) seja reservada e o seu endereço inicial seja retornado:

- isto é possível utilizando-se o operador **new**

#### Exemplo 1:

```
int * pa = new int; // new é um operador específico de C++;
                  // em C use a função malloc;
// Na linha acima, foi solicitado que seja reservado o espaço necessário para
// um valor do tipo int - e que seu endereço seja gravado na variável pa;
```

#### Exemplo 2:

```
int * pa = new int [5];
// Na linha acima, foi solicitado que seja reservado o espaço necessário para
// uma série de cinco valores do tipo int.
// E o endereço inicial dessa região é armazenado na variável pa;
```

✂ Quando **alocamos memória** no **heap** através de **new**, devemos também proceder à **liberação** dessa memória, usando:

- o operador **delete**, para liberar a memória alocada para um **único** elemento de um determinado tipo (“Exemplo 1”, acima).
- ou o operador **delete [ ]** (**delete** seguido de abre e fecha **colchetes**), para liberar a memória alocada para uma **série** de elementos de um determinado tipo (“Exemplo 2”, acima).

#### Exemplos:

```
int * pa = new int; // new é um operador específico de C++;
                  // em C use a função malloc;
```

```
.....
delete pa; // delete é um operador específico de C++;
           // em C use a função free;
```

OU

```
int * pa = new int [ 5 ];
.....
delete [ ] pa ; // neste caso foi usado “delete [ ]”
                // pois havia sido alocada memória para uma série de 5 int's.
```


Uma vez criada uma variável ponteiro e preenchida com um endereço de memória válido, podemos utilizá-la para acessar os valores armazenados na memória apontada.

Mas como deve ser feito esse acesso?

### 15.5.1.2 ▪ Acessando valores através de ponteiros.

Para acessar a região de memória cujo endereço está armazenado em uma variável ponteiro, precisamos também utilizar o operador ponteiro:

```
int * pa = new int ;
*pa = 5 ;
```

 Acima, o operador de ponteiro indica que **não** queremos escrever o número "5" em "pa".

- O que queremos é que "pa" seja acessada para que seja **lido o endereço nela armazenado**.
- Em seguida deve ser acessado **esse endereço** e, **aí sim**, escrever o número "5".


E, caso estivéssemos alocando memória para uma **lista** de inteiros (ao invés de um único "int", poderíamos utilizar também o operador ponteiro, indicando contudo **qual dos elementos da lista (um vetor, nesse caso)** queremos acessar.

 Isso pode ser feito com um **deslocamento a partir do endereço inicial**:

```
int * pa = new int [ 3 ];
*pa = 5 ; // foi acessado o primeiro elemento da série { *( pa + 0 ) = 5 ; }.
// abaixo, é acessado o segundo elemento da série:
*( pa + 1 ) = 10 ; // O "+1" aqui indica "mais um elemento desse tipo"
// e não "mais um byte".
// Como, no caso, temos um ponteiro para int, em um sistema
// de 32 bits teremos um deslocamento de quatro bytes
*( pa + 2 ) = 4 ; // acessa o terceiro elemento da série
```

E neste caso podemos também usar o operador de **índice [ ]** :

```
pa[ 0 ] = 5;
pa[ 1 ] = 10;
pa[ 2 ] = 4;
```

 Nos dois exemplos acima, alocamos memória para uma série de 3 ints.

- Em seguida utilizamos o endereço corretamente para acessar **cada um** dos três inteiros da série.
- Fizemos isso, no primeiro caso, através do operador **ponteiro**.
- E, no segundo caso, através do operador de **índice**.

### 15.5.1.3 ▪ Exercícios: demonstrando o funcionamento de ponteiros.

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

int VariavelExterna = 0 ;

int main()
{
    int Var = 5;

    int V2 = Var;
    /* Ler o conteúdo armazenado na memória apelidada de "Var"
    (que, no caso, é o número 5), e copiá-lo para "V2" ("V2" é uma variável do tipo int, isto é,
    ela está destinada a armazenar um valor inteiro)
    */
```

```
int * pVar = &Var ; /* Ler o endereço apelidado de "Var" e armazenar em "pVar" ("pVar"
é uma variável do tipo ponteiro para int,
isto é ela está destinada a armazenar endereços).
*/
```

```
int * Notas = new int[ 3 ] ; /* Endereço obtido no heap, com reserva de
memória para 3 valores inteiros, e armazenado na variável ponteiro Notas
*/
```

```
/*
```

*Abaixo, ERRO:*

```
int FalsoPonteiro = &Var; // FalsoPonteiro NÃO É um Ponteiro, logo
// não pode armazenar Endereços.
```

```
// Pois foi declarado como int e não como int *
```

```
*/
```

*// abaixo iremos imprimir as variáveis e seus endereços:*

```
cout << "Conteudo de Var = " << Var << endl ;
```

```
cout << "Conteudo de V2 = " << V2 << endl ;
```

```
cout << "Endereco de Var = " << (int)&Var << endl ;
```

```
cout << "Endereco de V2 = " << (int)&V2 << endl ;
```

```
cout << "Conteudo de pVar = " << (int) pVar << endl ;
```

```
cout << "Endereco de pVar = " << (int)&pVar << endl ;
```

```
cout << "Conteudo APONTADO por pVar = " << *pVar << endl ;
```

```
cout << "Endereco de Notas = " << (int) &Notas << endl ;
```

```
cout<<"Conteudo armazenado por Notas (endereço obtido no heap)="
<< ( int ) Notas << endl ;
```

```
cout << "Endereco de VariavelExterna (memoria global) = "
<< (int) &VariavelExterna << endl ;
```

```
delete [] Notas; // libera memória alocada no "heap"
```

```
return 0;
```

```
}
```

```
/*
```

RESULTADO (obs: os endereços podem variar de acordo com a plataforma):

```
Conteudo de Var = 5
```

```
Conteudo de V2 = 5
```

```
Endereco de Var = 6749684
```

```
Endereco de V2 = 6749680
```

```
Conteudo de pVar = 6749684
```

```
Endereco de pVar = 6749676
```

```
Conteudo APONTADO por pVar = 5
```

```
Endereco de Notas = 6749672
```

```
Conteudo armazenado por Notas (endereço obtido no heap) = 7933328
```

```
Endereco de VariavelExterna (memoria global) = 4382832
```

```
*/
```

Analisando o resultado:

- O endereço de Var corresponde ao conteúdo de pVar (que armazenou esse endereço).  
E pVar tem o seu próprio endereço.
- Observe também que as variáveis V2, Var e pVar e Notas têm os seus endereços bem próximos (essa é a faixa de memória da pilha).
- Já o endereço obtido no heap (ponteiro para o primeiro de três inteiros) e armazenado em Notas, está em outra faixa de memória: 7.933.328 (essa é a faixa do heap).
- E o endereço da variável global VariavelExterna está em uma terceira faixa: 4.382.832(essa é a faixa da memória global).

Segundo exercício. Percorrendo os elementos apontados por “Notas”.

```
#include <iostream>
int main()
{
    int * Notas = new int[ 3 ];
    cout << "Endereco obtido no heap e armazenado em Notas = "
          << (int) Notas << endl;

    int iC;
    for ( iC = 0 ; iC < 3 ; iC++ )
    {
        Notas[ iC ] = iC+1 ;
        cout      << "Elemento " << iC
                  << "\tValor = "      << Notas[ iC ]
                  << "\tEndereco = "  << int( Notas + iC )
                  << endl;
    }
    return 0;
}
/*
RESULTADO (em plataforma de 32 bit's):
```

Endereco obtido no heap e armazenado em Notas = 7933328

Elemento 0	Valor = 1	Endereco = 7933328
Elemento 1	Valor = 2	Endereco = 7933332
Elemento 2	Valor = 3	Endereco = 7933336

\*/

Analisando o resultado.

- O endereço armazenado em “Notas” é o endereço do primeiro elemento da série.



- O endereço dos demais elementos é igual ao endereço do anterior mais quatro bytes (tamanho de um int em 32 bit's) .
- Imprimimos os valores de cada elemento da série usando o operador de índice, com o devido deslocamento:

Notas[ iC ]

Mas também atingiríamos o mesmo resultado usando o operador ponteiro:

\*( Notas + iC )

- Imprimimos os endereços de cada elemento da série simplesmente somando o deslocamento ao endereço inicial:

Notas + iC

## 15.5.2 • Ponteiros inteligentes (smart pointers)

Uma memória alocada manualmente através do operador **new** deve ser liberada manualmente através de **delete**. Se isso não for feito, a memória permanecerá alocada até que o programa termine sua execução. Isso pode ser muito perigoso em determinadas situações:

### Situação 1 – Return antes do delete

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

struct Cliente
{
    string nome;
    Cliente()
    {
        cout << "Construindo...\n\n";
    }
    ~Cliente()
    {
        cout << "\n\nDestruido...\n";
    }
};

int main()
{
    Cliente *cliente = new Cliente;
    cout << "Digite seu nome: ";
    cin >> cliente->nome;

    if(cliente->nome.length() < 6)
    {
        cout << "Nome invalido.\n";
        cout << "O programa será encerrado.\n";
        return 0;
    }
    cout << "Ola, " << cliente->nome << '\n';
    delete cliente;
```

```
    return 0;
}
```

Neste exemplo, se o nome possuir mais de 5 caracteres a memória “cliente” será liberada, mas se o nome possuir menos de 6 caracteres, haverá um vazamento de memória (memory leak), pois existe um **return** para essa condição.

## Situação 2 – Exceções antes do delete

```
#include <iostream>
#include <stdexcept>

using namespace std;

struct Cliente
{
    string nome;
    Cliente()
    {
        cout << "Construindo...\n\n";
    }
    ~Cliente()
    {
        cout << "\n\nDestruindo...\n";
    }
};

void imprimir(const Cliente &cliente)
{
    if(cliente.nome.length() < 6)
    {
        length_error erro("Nome invalido");
        throw erro;
    }
    cout << "Ola, " << cliente.nome << '\n';
}

int main()
{
    Cliente *cliente = new Cliente;
    cout << "Digite seu nome: ";
    cin >> cliente->nome;

    imprimir(*cliente);

    delete cliente;

    return 0;
}
```

Se o nome possuir mais de 5 caracteres a memória “cliente” será liberada, mas se o nome possuir menos de 6 caracteres, haverá um vazamento de memória (memory leak), pois uma exceção será emitida.

### 15.5.2.1 ▪ Auto ptr (obsoleto a partir de C++11)

Para solucionar os problemas descritos acima, havia o `auto_ptr`, onde era feita uma alocação de memória com **new** em seu construtor e o **delete** era feito automaticamente em seu destrutor.

```
#include <iostream>
#include <memory>

using namespace std;

struct Cliente
{
    string nome;
    Cliente()
    {
        cout << "Construindo...\n\n";
    }
    ~Cliente()
    {
        cout << "\n\nDestruindo...\n";
    }
};

int main()
{
    auto_ptr<Cliente> cliente(new Cliente);
    cout << "Digite seu nome: ";
    cin >> cliente->nome;

    if(cliente->nome.length() < 6)
    {
        cout << "Nome invalido.\n";
        cout << "O programa será encerrado.\n";
        return 0;
    }
    cout << "Ola, " << cliente->nome << '\n';

    return 0;
}
```

A classe `std::auto_ptr` possuía algumas limitações, como, por exemplo, permitir que um objeto `std::auto_ptr` copie a memória do outro e a apague sem notificá-lo e não permitir a criação de arrays. Por esses motivos `std::auto_ptr` é considerado obsoleto no padrão C++11, sendo substituído por `std::weak_ptr`.

### 15.5.2.2 ▪ weak\_ptr (C++11)

`std::weak_ptr` é o sucessor de `std::auto_ptr`, mas possui várias melhorias, como por exemplo, pode armazenar ponteiros para arrays.

**OBS:** Um `std::weak_ptr` não pode ser copiado, somente movido.

```
#include <iostream>
#include <memory>
```

```
using namespace std;

int main()
{
    //Cria um unique_ptr que instancia um int no heap
    unique_ptr<int> u(new int(123));

    cout << *u << '\n';

    //Move o conteudo do ponteiro de u para u2
    unique_ptr<int> u2(move(u));

    //Erro, porque o conteudo de um foi movido para u2. u é nulo
    //cout << *u << '\n';

    //unique_ptr<int> u3(i); Erro pois nao permite copia

    //Aloca um vetor de 10 inteiros no heap
    unique_ptr<int []> uv(new int[10]);

    //Insera dados no vetor
    for(int i = 0; i < 10; ++i)
        uv[i] = i;

    //Exibe os dados
    for(int i = 0; i < 10; ++i)
        cout << uv[i] << '\n';

    return 0;
}
```

### 15.5.2.3 • shared\_ptr (C++11)

Permitem compartilhar uma memória alocada por N ponteiros e guarda um contador de referências que é incrementado em 1 a cada novo ponteiro que compartilhar a mesma memória.

Quando o contador chegar a 0 a memória será desalocada automaticamente.

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

int main()
{
    shared_ptr<int> pI(new int(0));
    cout << "Contador: " << pI.use_count() << "\n\n";

    {
        shared_ptr<int> pI_2(pI);
        cout << "Contador: " << pI_2.use_count() << "\n\n";
    }

    cout << "Contador: " << pI.use_count() << "\n\n";
    return 0;
}
```

### 15.5.2.4 ▪ weak\_ptr (C++11)

Usado em conjunto com um `std::shared_ptr`, pode dizer se o ponteiro continua ou não válido. Quando o contador de um `std::shared_ptr` chegar a zero, todos os `std::weak_ptr` relacionados a eles expiram.

```
#include <iostream>
#include <memory>

using namespace std;

int main()
{
    shared_ptr<int> pI_shared(new int(0));
    cout << "Contador: " << pI_shared.use_count() << '\n';

    //Não aumenta o contador weak_ptr<int> pI_weak(pI_shared);

    cout << "Contador: " << pI_shared.use_count() << '\n';

    //Verifica se o ponteiro não esta expirado
    if(!pI_weak.expired())
        cout << "Nao expirado!!!\n";
    else
        cout << "Expirado!!!\n";

    //Expira o ponteiro
    cout << "Contador: " << pI_shared.use_count() << '\n';

    pI_shared.reset();

    //Verifica se o ponteiro não esta expirado
    if(!pI_weak.expired())
        cout << "Nao expirado!!!\n";
    else
        cout << "Expirado!!!\n";

    return 0;
}
```

Um `std::weak_ptr` não pode na verdade ser usado como um smart pointer, pois, não possui os operadores de `*` e `->` que permitem o acesso ao ponteiro. Para usar um `std::weak_ptr`, temos que criar um `std::shared_ptr` a partir dele.

Se o `std::weak_ptr` estiver inválido, então o `std::shared_ptr` irá conter um ponteiro nulo.

```
#include <iostream>
#include <memory>

using namespace std;

int main ()
{
    //Apenas cria shared_ptr, sem instanciar nada no heap
```

```

shared_ptr<int> sp1, sp2;

//Apenas cria shared_ptr, sem apontar pra nenhum shared_ptr
weak_ptr<int> wp;

//Instancia um int no heap e faz sp1 apontar para esse int
sp1 = std::make_shared<int> (20);

//agora weak_ptr wp aponta para sp1
wp = sp1;

//Retorna o shared_ptr
sp2 = wp.lock();

//Reseta o ponteiro
sp1.reset();

sp1 = wp.lock(); //Põe novamente o int alocado no heap

std::cout << "*sp1: " << *sp1 << '\n';
std::cout << "*sp2: " << *sp2 << '\n';

return 0;
}

```

### 15.5.3 • Vetores e Matrizes.

Nos exemplos mostrados acima, verificamos que podemos utilizar ponteiros tanto para armazenar o endereço de uma única variável como também para armazenar o endereço inicial de uma **lista de dados**(ou seja, o endereço do primeiro elemento da lista).

Uma lista de dados é um **vetor**. Vetores são uma das aplicações mais importantes de ponteiros. Se temos uma série de dados **do mesmo tipo** (uma lista de inteiros, por exemplo), precisaremos de um ponteiro para guardar o endereço do **primeiro elemento da lista** (o qual, por sua vez, será endereçado como elemento zero do vetor).

Agora podemos navegar fácil e rapidamente por toda a lista, pois, como todos os elementos da lista têm o mesmo tamanho (porque são do mesmo tipo) tanto o operador de ponteiro como o operador de índice poderão atingir qualquer elemento com uma fórmula de acesso simples.

- Quando escrevemos:
 

```
int * pa = new int [ 5 ];
```

  - E em seguida acessamos um elemento do vetor:
 

```
*( pa + 3 ) = 4;
```
  - ou
 

```
pa [ 3 ] = 4;
```
- Esse código é realizado do seguinte modo:
 

```
pa + ( 3 * sizeof( int ) ) = 4;
```

- Ou seja: é feito um deslocamento, a partir do endereço inicial, de modo a atingir o endereço do elemento do vetor que precisamos acessar.
- Isto significa que tanto o operador de ponteiro como o operador de índice, incorporam uma **aritmética de ponteiro**.
- Assim, **pa + 1**, por exemplo, **não** significa “pa mais 1 byte” e sim “pa mais o número de bytes especificado pelo seu **tipo**”.

### 15.5.3.1 ▪ Onde alocar vetores: memória global, pilha ou *heap* ?

Podemos alocar vetores no “*heap*”, na pilha ou na memória global.

```
int * p1 = new int [ 5 ] ; // Vetor alocado no “heap”, através
                          // do operador de livre alocação.

{
    int p2 [ 5 ] ; // Declarada dentro de um bloco, sem especificação static;
                  // então a variável é da classe auto por default.
                  // Logo, o vetor será alocado na pilha.
}

int p3 [ 5 ] ; // Declarada fora de bloco, sem especificação static: a variável é da classe global.
              // Logo, o vetor será alocado na memória global.

static int p4 [ 5 ] ; // Declarada fora de bloco, com especificação static.
                     // Então a variável é da classe static, externa.
                     // Logo, o vetor será alocado também na memória global.
```



A faixa de memória **global**, portanto, serve para abrigar todas as variáveis com **tempo de vida** global, isto é, com o mesmo tempo de vida da aplicação.


E qual é a diferença entre alocar uma variável na pilha ou na memória global **ao invés** de alocá-la no “*heap*” ?

- A memória **global** tem um tamanho **fixo**.
- Esse tamanho é determinado pela soma de todas as variáveis cujo **tempo de vida** é o mesmo da aplicação (variáveis globais e estáticas).
- E esse tamanho já é conhecido e durante a compilação e *link-edição*.
- É então fixado e **não pode** mais ser **alterado**.
- A **pilha** também tem um tamanho **fixo**.
- Ele deve ser estipulado antes de procedermos à *link-edição*.
- Podemos alterar o tamanho da pilha nas opções do *linker*. Contudo, uma vez gerado o executável **não será mais possível alterar** o tamanho da pilha.
- O tamanho da pilha pode assim ser determinado pelo programador, que deveria definir um tamanho não muito pequeno (do contrário, teríamos um estouro de pilha), mas também não muito grande (pois isto reduz a eficiência no gerenciamento da pilha, reduzindo performance).
- Os *link-editores* normalmente já definem um tamanho de pilha *default* levando em conta a plataforma (e normalmente esse é o melhor tamanho).
- Mas qualquer alteração tem que ser feita **antes** da *link-edição* e não em tempo de execução.
- **Conclusão:**
  - **Tanto** na memória **global** **como** na **pilha** **não podemos** utilizar memórias de tamanho **variável** ou que devam ser **alteradas dinamicamente**, em tempo de execução.
  - Se queremos que um vetor tenha uma **dimensão desconhecida** em tempo de compilação (pois o seu tamanho só será conhecido já

em execução), ou se queremos **redimensionar** esse vetor, então o **local apropriado é o heap**.

- Essa distinção entre memória fixa e memória livre é um dos fatores que visam garantir **performance** no uso da memória.

---

 Portanto, ao alocar um vetor **no heap não** precisamos conhecer previamente a quantidade de elementos.

---

E, assim sendo, podemos fazer:

```
int x ;
..... // a variável x receberá um valor qualquer
int * pa = new int [ x ] ; // é reservada memória no heap para x inteiros;
                        // o endereço inicial é retornado por new e armazenado em pa;
```

E, igualmente, podemos redimensionar o vetor:

```
.....
delete [ ] pa ; // a memória reservada no heap para x inteiros é liberada;
pa = new int[ 10 ] ; // é reservada memória no heap para 10 inteiros;
                    // um novo endereço inicial é retornado por new e armazenado em pa;
.....
delete [ ] pa ; // é liberada a memória para 10 inteiros reservada no heap
int y ;
..... // a variável y receberá um valor qualquer
pa = new int [ y ] ; // é reservada memória no heap para y inteiros;
                    // um novo endereço inicial é retornado por new e armazenado em pa;
.....
delete [ ] pa ; // a memória reservada no heap para y inteiros é liberada;
```

Já se o vetor tivesse sido declarado na pilha, ou na memória estática, não poderíamos utilizar uma quantidade variável de elementos(desconhecida em tempo de compilação).

E também não poderíamos redimensionar o vetor.

### Exemplos:

```
{
    int iVetorNaPilha[ 20 ] ; // vetor de 20 elementos declarada na pilha;
    // A quantidade de elementos é constante,
    // e portanto conhecida em tempo de compilação.
    // O vetor não poderá ser redimensionado.
}

int iVetorNaMemoriaEstatica[ 30 ] ; // vetor de 20 elementos declarada na
// memória global;
// A quantidade de elementos é constante,
// e portanto conhecida em tempo de compilação.
// Também neste caso, o vetor não poderá ser redimensionado.
```

Além da distinção básica entre tamanho fixo e tamanho variável, há um outro elemento que devemos levar em conta:





Vetores muito **grandes não** devem ser alocadas na **pilha**, justamente porque o seu tamanho é arbitrado antes da link-edição. Assim um vetor muito grande ocasionará um **estouro de pilha**.

Por isso, vetores muito grandes ou são alocados na memória global (se forem necessários durante **todo** o tempo de vida da aplicação) ou, **preferencialmente, no heap**



Mas **evite** usar **new/delete** diretamente.

Ao invés disso, use `std::vector`, `std::string` e outros **containers da STL** para conjunto de dados.

E para um único objeto alocado na memória dinâmica, use **`std::unique_ptr` ou `std::shared_ptr`**.

### 15.5.3.2 • Matrizes - usando múltiplas dimensões.

A linguagem não estabelece um limite para a quantidade de dimensões para uma matriz. Na prática contudo esse limite será estabelecido pela memória disponível.

Assim podemos fazer:

```
#include <iostream>

int main()
{
    double PlanilhaCalculos[ 6 ][ 7 ]; // matriz de duas dimensões
    // representando uma planilha de cálculos de 6 linhas e 7 colunas
    // escrever 9.8 na "Linha 1", "Coluna 1":
    PlanilhaCalculos[ 0 ][ 0 ] = 9.8; // matriz é indexada a partir de zero
    // escrever 10.3 na "Linha 5", "Coluna 4":
    PlanilhaCalculos[ 4 ][ 3 ] = 10.3; // matriz é indexada a partir de zero
```

/\* RESULTADO:

	0	1	2	3	4	5	6
0	9.8						
1							
2							
3							
4				10.3			
5							

\*/

```
cout << PlanilhaCalculos[ 0 ][ 0 ] << endl;
cout << PlanilhaCalculos[ 4 ][ 3 ] << endl;
```

```

        /* e será impresso:
                                9.8
                                10.3
        */
    return 0;
}

```

### 15.5.3.3 ▪ Inicialização de vetores e matrizes.

A regra para inicialização (atribuição na própria linha de declaração) de matrizes prevê que seja apresentada uma lista de valores delimitada por chaves. Não é necessário contudo inicializar todos os elementos da matriz:

```

// A) inicializando todos os elementos de uma matriz:
int MatrizDeInt[3] = { 1 , 4, 2 } ;

// B) inicializando apenas alguns elementos de uma matriz:
int OutraMatrizDeInt[3] = { 1 , 4 } ;

```

🔗 Se uma matriz for **inicializada**, podemos **omitir** a dimensão explícita. Neste caso o compilador se baseará na **quantidade** de elementos **inicializados** e usará esse número constante como dimensão da matriz:

```

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

// A Matriz acima será dimensionada para quatro elementos, já que a
// dimensão explícita foi omitida e foram inicializados quatro elementos.

```

### 15.5.3.4 ▪ Vetores e Matrizes de estruturas.

Uma matriz de estruturas não difere de uma matriz para um tipo primitivo. Basta combinar a lógica de matrizes e a lógica de estruturas:

```

struct Area
{
    public:
        int Largura;
        int Altura;
};

int main ( )
{
    Area MatrizDeAreas [ 10 ]; // uma matriz para dez variáveis do tipo "Area"
    // agora vai acessar os dois campos do quinto elemento da matriz:

    MatrizDeAreas[ 4 ].Largura = 10;
    MatrizDeAreas[ 4 ].Altura  = 20;

    return 0;
}

```

### 15.5.3.4.a ▪ Inicializando uma matriz de estruturas

Em primeiro lugar, podemos **inicializar uma estrutura** também utilizando as chaves delimitadoras para definir um valor para cada campo da estrutura.

Por exemplo:

```
struct Area
{
    public:
        int Largura;
        int Altura;
};

int main ( )
{
    Area MinhaArea = { 10 , 15 } ;
        // Acima, foi inicializada a variável estruturada "MinhaArea",
        // com a atribuição do valor 10 para o campo Largura
        // e 15 para o campo Altura

    return 0 ;
}
```

Observe que a inicialização de uma estrutura, embora em ocasiões muito raras possa ser útil, apresenta um problema grave: a inicialização é baseada na **ordem de declaração dos campos e não nos seus nomes**.

Assim, se essa ordem for modificada algum dia (ou um novo campo for acrescentado, mudando a ordem) todas as inicializações feitas no passado ficarão incorretas e terão que ser alteradas.



Além disso a **struct** ou **class C++** **não poderá** ser inicializada com esse método **se** uma **construtora** for declarada explicitamente. Neste caso a inicialização deverá ser feita **na própria construtora**.

De todo modo, vejamos como seria a inicialização de uma matriz de estruturas, **combinando** a regra de inicialização de uma **matriz** com a regra de inicialização de uma **struct**:

```
int main( )
{
    Area MinhaArea [3] =
        { // abre a chave para inicializar a MATRIZ

        // abaixo: vai inicializar cada elemento da matriz, isto é, cada variável
        // estruturada, também abrindo e fechando chaves para cada um.
        // Os três pares de chaves abaixo, portanto, referem-se à inicialização
        // dos campos da estrutura.

        { 10 , 20 }, // largura e altura para o primeiro elemento da matriz
        { 15 , 9 }, // largura e altura para o segundo elemento da matriz
        { 6 , 8 } // idem para o terceiro e último elemento

    } ; // fecha a chave de inicialização da MATRIZ

    return 0 ;
}
```

### 15.5.3.5 • Vetores e Matrizes de caracteres.

Podemos usar o tipo inteiro **char** (um byte) para representar um caracter. Isto porque o char pode ser **relacionado** à tabela de caracteres da máquina (tabela **ASC**, no caso do IBM-PC).

Assim podemos fazer:

```
char Letra_A = 65 ;
```

OU

```
char Letra_A = 'A' ; // caracter A delimitado com aspas simples
```

O valor armazenado em ambos os casos será sempre o número 65 em sua representação binária.

Assim se fizermos:

```
char Letra_A = 'A' ;
char Letra_B = 'B' ;
cout << Letra_A + Letra_B << endl ;
```

OU

```
cout << 'A' + 'B' << endl ;
```

Em ambos os casos o resultado é 131 ( 65 + 66 ).

Para fins de impressão e visualização humana, a exibição do número será feita através do caracter correspondente na tabela de caracteres. Isso permitirá imprimir textos.

Em **C** ou **C++**, podemos compor um texto através de uma **série** de caracteres, e isto pode ser implementado como uma **matriz de caracteres**.

E, para facilitar o trabalho com textos, a linguagem oferece duas **convenções específicas**, válidas apenas para matrizes de caracteres:

a) as **aspas duplas**.

- Uma matriz de caracteres (ou uma **string**) constante pode ser representada com **aspas duplas**, como está abaixo:

```
"JOSE DA SILVA"
```

b) o **terminador zero**.

- Em um texto normal o zero binário não é utilizado (pois não tem qualquer representação na tabela de caracteres).
- Por isso, por convenção, ele é empregado para indicar "**fim de texto**".
- Desse modo se declararmos uma matriz de caracteres com dimensão 30, mas utilizarmos apenas as 3 primeiras posições (por exemplo, com o nome "IVO"), o quarto caracter será preenchido com o zero binário (pois as aspas duplas sempre reservam um byte a mais e o preenchem com o zero binário).
- Se, em seguida, utilizarmos **printf** ou **cout**, a convenção será respeitada e será impressa apenas a palavra "IVO", não sendo impresso o "lixo" posterior ao terminador zero.
- O terminador zero pode ser **representado** de duas maneiras:
- em formato hexadecimal : **0x0**.
- O "0x" antes de um número indica que esse número usa a base hexadecimal e não decimal
- na representação caracter, entre aspas simples e precedido pelo caracter de controle '\': **'\0'**

Para uma grande quantidade de casos (texto propriamente dito) o uso do terminador é sem dúvida o modo mais eficiente e simples de indicar o fim de texto.

Contudo, encontraremos **algumas situações** em que **não poderemos** considerar o zero binário como sendo um **terminador**.

Por exemplo, em um texto criptografado o zero binário poderá aparecer diversas vezes. E, neste caso, ele não estará indicando o fim do texto e sim o resultado da criptografia sobre o caracter original. Nenhum algoritmo eficiente de criptografia pode proibir o uso de qualquer valor possível (inclusive o zero), pois do contrário no momento de descriptografar teríamos uma grande complicação.

E há outras situações desse tipo. Nesses casos, teremos que utilizar o controle do fim de texto por **tamanho** e não por um terminador. Isto é: teremos que reservar alguns bytes extras para armazenar um número inteiro que indique quantos caracteres estão sendo efetivamente usados em uma cadeia de caracteres.

### 15.5.3.5.a • Inicializando uma matriz de caracteres

Uma matriz de caracteres pode ser inicializada das seguintes maneiras:

- 1) usando a forma geral de inicialização de matrizes:

```
char String_1 [ 4 ] = { 65 , 66 , 67 , 0x0 } ;
char String_2 [ 4 ] = { 'A' , 'B' , 'C' , '\0' } ;
char String_3 [ ] = { 'A' , 'B' , 'C' , 0x0 } ; //dimensão = 4
```

- 2) usando as convenções arbitrárias para matriz de caracteres (aspas duplas com terminador zero embutido):

```
char String_4 [ 4 ] = "ABC" ;
char String_5 [ ] = "ABC" ; // dimensão = 4
```

Se agora fizermos:

```
cout << String_1 << endl;
cout << String_2 << endl;
cout << String_3 << endl;
cout << String_4 << endl;
cout << String_5 << endl;
```

*Em TODOS os casos será impresso: ABC*

### 15.5.3.5.b • Entendendo matriz de caracteres como um endereço

Como ocorre com **qualquer** matriz, uma variável matriz de caracteres também representa o **endereço do primeiro elemento da lista**. A partir desse endereço podemos nos deslocar, atingindo os demais elementos.

```
char Nome[ 30 ] = "MARIA" ;
Nome[ 3 ] = 'T' ; //aspas simples pois é um único char.
// Se agora imprimirmos a variável Nome, o resultado será: MARTA
```

```
char Nome[ ] = "Jose da Silva" ;
cout << Nome + 5 << endl ;
// Será impresso: da Silva
```

```
cout << "IVO MOREIRA" + 4 << endl ;
// Será impresso: MOREIRA
```

### 15.5.3.5.c • Exemplo de matriz de caracteres

Vamos ver agora um exemplo mais completo. Nele utilizaremos duas das funções da biblioteca padrão do C: **strcpy**, **strncpy** e **strlen**;

- **strcpy** copia uma matriz de caracteres para outra (o que significa uma cópia a partir de um endereço para um outro endereço);
- já **strncpy** também copia, mas acrescenta um parâmetro numérico para permitir que seja copiada apenas uma quantidade máxima de bytes, caso o terminador zero não seja encontrado antes;
- e **strlen** retorna quantos bytes existem **antes** do terminador zero.

Além disso vamos escrever uma função (StrCopia) semelhante a strncpy. Essa função irá fazer basicamente a mesma coisa, mas diferentemente de strncpy, ela garante sempre um terminador zero após o último caracter copiado.

```
#include <iostream>
#include <string>

// função para copiar todos os elementos de uma matriz de caracteres para outra:
int StrCopia ( char * sDestino,  const char * sOrigem, int nMaxBytes );

int main()
{
    // Abaixo: reserva de 31 bytes; no momento estão sendo usados 6
    // ( [ m, a, r, i, a ] mais o terminador zero)
    char Nome1 [ 31 ] = "Maria";
    cout << Nome1 << endl;  // imprime: Maria

    // abaixo, ERRO:
    // char Nome[5] = "Maria"; // faltou reserva para o terminador zero
    // ( no mínimo deveria ser feito : Nome[ 6 ] = "Maria"; )

    // reserva de 6 bytes, todos em uso:
    char Nome2[ ] = "Pedro";
    cout << Nome2 << endl;  // imprime: Pedro

    char Nome3[31] = { 'M', 'A', 'R', 'I', 'A', '\0' };
    cout << Nome3 << endl;  // imprime: MARIA

    strcpy(Nome1, "Jose da Silva");
    cout << Nome1 << endl;  // Imprime : Jose da Silva

    // Abaixo, ERRO: Nome2 tem uma reserva para apenas 6 bytes:
    // strcpy(Nome2, "Jose da Silva"); // "Jose da Silva" tem 13 bytes

    // agora vamos usar strncpy, com controle de cópia por tamanho máximo:
    strncpy(Nome3, "Carlos Lopes Costa", 50 );
    cout << Nome3 << endl;  // Imprime : Carlos Lopes Costa.
    // Antes de atingir 50 caracteres strncpy encontrou o terminador zero

    // Agora vamos usar a função StrCopia,
    // implementada mais abaixo

    StrCopia(Nome1, "Antonio da Costa", 30);
    cout << Nome1 << endl;  // Imprime: Antonio da Costa

    StrCopia(Nome1, "Marcos Albuquerque Medeiros e Silva", 30);
    cout << Nome1 << endl;  // Imprime: Marcos Albuquerque Medeiros e

    // simulando uma função "Left"
    StrCopia(Nome1, "Joao da Silva", 4);
    cout << "Left: " << Nome1 << endl;  // Imprime: Left: Joao

    // simulando uma função "Mid" ou "SubString"
```

```

StrCopia(Nome1, Nome3 + 7, 5 );
cout << "Mid: " << Nome1 << endl; // Imprime : Mid:Lopes
StrCopia ( Nome1 , "Jose Leonardo Rocha" + 5 , 8 );
cout << "Mid: " << Nome1 << endl; // Imprime: Mid:Leonardo
// simulando uma função "right"
StrCopia(Nome1, Nome3 + strlen(Nome3) - 5 , 5 );
cout << "Right: " << Nome1 << endl; // Imprime: Right:Costa
return 0;
}

// função para copiar todos os elementos de uma matriz de caracteres para outra:
int StrCopia ( char * sDestino , const char * sOrigem , int nMaxBytes )
{
    int nByteDaVez = 0;
    while ( sOrigem[nByteDaVez] != '\0' && nByteDaVez < nMaxBytes )
    {
        sDestino[nByteDaVez] = sOrigem[nByteDaVez];
        // outra forma de escrita:
        //*(sDestino+nByteDaVez) = *(sOrigem+nByteDaVez);
        nByteDaVez++;
    }
    sDestino[nByteDaVez] = '\0';
    return nByteDaVez; //retorna a quantidade de bytes efetivamente copiados.
}

//OUTRA FORMA de escrever a função StrCopia.
// Obs: Esta forma é mais eficiente (maior velocidade de execução).
int StrCopia ( char * sDestino , const char * sOrigem , int nMaxBytes )
{
    const char * sOrigemInicial = sOrigem;
    while ( *sOrigem != '\0' && sOrigem - sOrigemInicial < nMaxBytes )
    {
        *sDestino = *sOrigem ;
        sOrigem++ ; sDestino++;
    }
    *sDestino = '\0';
    return sOrigem - sOrigemInicial; //retorna: quantidade de bytes copiados.
}

// Outra opção: usando o laço for:
int StrCopia ( char * sDestino, const char * sOrigem, int nMaxBytes )
{
    const char * sOrigemInicial ;
    for ( sOrigemInicial = sOrigem ;
        *sOrigem != '\0' && sOrigem - sOrigemInicial < nMaxBytes ;
        sOrigem++ , sDestino++ )
    {
        *sDestino = *sOrigem ;
    }
    *sDestino = '\0' ;
    return sOrigem - sOrigemInicial;
    //retorna: quantidade de bytes copiados.
}

```


## 15.6 • Ponteiros para funções(exemplo adicional)

### A estrutura relatório.


Um bom exemplo do uso de ponteiros para função seria a impressão de relatórios. De um modo geral, diferentes relatórios apresentam características comuns: todos podem ter um cabeçalho e todos devem ter linhas de detalhe, por exemplo. Além desses atributos comuns, existe uma **lógica comum**:

- É preciso sempre checar se o total de linhas que cabem em uma folha já foi impresso.
- Se isso ocorreu deve-se saltar de página.
- Em seguida, na maioria dos casos, é preciso imprimir um cabeçalho.
- Agora, devem ser impressas as linhas de detalhe.
- etc.

---

 Desse modo, podemos escrever uma camada de código genérico onde essa lógica fique pronta.

---

 Mas, evidentemente, o código genérico não saberá como imprimir as **linhas de detalhe**, pois este é o aspecto de um relatório que é sempre específico.

---

 Então, **nesse ponto** da sequência de operações, chamamos uma função **a partir de um ponteiro**.

---

Assim, quem for usar esse código genérico poderá preencher esse ponteiro com o endereço de uma função de impressão específica.

Para cada relatório diferente escrevemos uma função específica que será responsável pela impressão das linhas de detalhe daquele relatório.

Em seguida preenchemos o ponteiro para função da estrutura de relatório com o endereço **dessa** função.

Podemos fazer coisa semelhante também para o cabeçalho.

No caso do cabeçalho, o código genérico já pode oferecer uma solução pronta, imprimindo uma ou mais linhas de cabeçalho informadas como parâmetro.

Mas ele poderia oferecer também um ponteiro para função, que, se estiver preenchido com o endereço de uma função específica, permitirá **modificar** o modo padrão de imprimir o cabeçalho. E, caso o ponteiro esteja nulo, seria usado simplesmente o cabeçalho padrão.

Veja, abaixo, o exemplo de um relatório padrão. Ele está implementado de forma bem simplificada. Mais a frente teremos oportunidade de ver uma outra versão desse mesmo exemplo implementada de modo mais detalhado.

```
#include <fstream.h>
#include <string.h>
#include <stdio.h>

// tipo de ponteiro para função
typedef bool ( * tpfRelatImpr )();

struct RelatPadrao
{
    private: // ATRIBUTOS:
        char * m_psCabeçalho; // string cabeçalho
        short m_nContaFolhas; // folhas impressas
        short m_nContaLinhas; //linhas impressas:
        short m_nLinhasPorFolha; // linhas por Folha
```



```

public:
    ofstream m_osSaida; //objeto para imprimir
public: // MÉTODOS
    RelatPadrao(char * psCabecalho);
    bool Imprime( char * sSaida );
    void ImprCabecalho();
    // ponteiros para funções:
    tpfRelatImpr m_pfCabec ;
    tpfRelatImpr m_pfDetalhe;
};
RelatPadrao::RelatPadrao(char * psCabecalho)
{
    // armazena strings para cabeçalhos:
    m_psCabecalho = psCabecalho;
    m_nLinhasPorFolha = 58 ;
    m_pfCabec = NULL;
    m_pfDetalhe = NULL;
}
void RelatPadrao::ImprCabecalho()
{
    if ( m_psCabecalho == NULL ) return ;
    if(m_nContaFolhas > 1)
        m_osSaida << "\f\n" ; // SALTO DE PÁGINA
    // posição para número de folha deve começar com
    // um '#' e terminar com um '#'
    char * pNum = strchr(m_psCabecalho, '#');
    char * pFimNum;
    if ( pNum != NULL )
    {
        *pNum = ' ';
        pFimNum = strchr(pNum + 1 , '#');
        int nQuantosNumeros = (pFimNum==NULL)
            ? 0 : pFimNum- pNum - 1;
        if ( nQuantosNumeros > 0 )
        {
            sprintf(pNum+1, "%0*d",
                nQuantosNumeros, m_nContaFolhas);
            *pFimNum = ' ';
        }
    }
    // imprime cabeçalho já com número da folha:
    m_osSaida << m_psCabecalho;
    // imprime uma linha separadora em branco:
    m_osSaida << '\n';
    if ( pNum != NULL ) // restaura '#'
    {
        *pNum = '#';
        if ( pFimNum != NULL )
            *pFimNum = '#';
    }
}
bool RelatPadrao::Imprime(char * sSaida)
{
    //se o ponteiro para a função que imprime a linha
    //de detalhe estiver NULO, então nada a fazer:
    if ( m_pfDetalhe == NULL )
        return false;
    m_osSaida.open(sSaida);

```

```

m_nContaFolhas = 0;
m_nContaLinhas = m_nLinhasPorFolha + 1;
bool bOk = m_osSaida.is_open() ;
while ( bOk )
{
    m_nContaLinhas++;
    if( m_nContaLinhas > m_nLinhasPorFolha )
    {
        ++m_nContaFolhas ;
        // se o ponteiro m_pfCabec estiver NULO, OU se // contiver o
        endereço de uma função e esta
        // retornar true, então imprime o cabeçalho.
        if( m_pfCabec == NULL || ➡ m_pfCabec() )
            ImprCabecalho();
        m_nContaLinhas = 3;
    }
    //AQUI CHAMA O PONTEIRO PARA IMPRIMIR DETALHES:
    ➡ ➡ bOk = m_pfDetalhe();
} //Fim do laço: encerrará quando bOK se tornar falso.
if ( m_osSaida.is_open() )
{
    m_osSaida << "\f\n" ; // Salto de página
    m_osSaida.close() ; //Fecha saída
}
return bOk;
}

➡ // Agora vamos usar a estrutura de relatório.
char Titulo[] = "Titulo Relatorio - Folha =#000#\n";
RelatPadrao Rel( Titulo ); //objeto RelatPadrao;
static int nLinhaTeste=0;
// função para impressão da linha de detalhe. Seu endereço // deverá ser atribuído ao
ponteiro membro da classe
➡ bool ImprimeDetalhes()
{
    nLinhaTeste ++;
    // Imprime a linha de detalhe:
    Rel.m_osSaida << "Linha = "
                    << nLinhaTeste << '\n';
    // quando atingir 120 linhas retorna falso,
    // finalizando o relatório:
    return ( nLinhaTeste <= 119 );
}
int main() // função main
{
    // alimenta o ponteiro para função
    ➡ Rel.m_pfDetalhe = ImprimeDetalhes;
    Rel.Imprime("Saida.prn");
    // se quiser enviar para a impressora,
    // troque a linha acima para:
    // Rel.Imprime( "LPT1" )
    return 0;
}

```



---

## • Anexos

---

<b>Anexo A.</b>	Compilador e <i>linker</i> .....	422
-----------------	----------------------------------	-----

<b>Anexo B.</b>	Respostas às questões para revisão.....	424
-----------------	---	-----

1.	Questões do Capítulo 1.....	425
2.	Questões do Capítulo 2.....	427
1.	Exercício.....	427
2.	Questões.....	428
3.	Questões do Capítulo 3.....	431
1.	Exercício.....	431
2.	Questões.....	433
4.	Questões do Capítulo 4.....	436
1.	Exercício.....	436
2.	Questões.....	440
5.	Questões do Capítulo 5.....	446
1.	Exercício 1.....	446
2.	Exercício 2.....	452
3.	Questões.....	455
6.	Questões do Capítulo 6.....	468
7.	Questões do Capítulo 7.....	471
8.	Questões do Capítulo 11.....	473

<b>Anexo C.</b>	Guia de consulta rápida.....	476
-----------------	------------------------------	-----

1.	Tabela de tipos primitivos.....	477
2.	Tabela de operadores <i>básicos</i> por tipo de operação.....	479
3.	Tabela completa de operadores e suas precedências.....	481
4.	Controles de fluxo de processamento.....	483
5.	Principais diretivas de compilação.....	485
6.	sequências <i>escape</i> .....	486
7.	Palavras reservadas.....	486
8.	Modificadores.....	487



## Anexo A. Compilador e *linker*

O **compilador** gera código de máquina (arquivos objeto: .obj ou .o), mas isso ainda não pode ser executado.

Pois o compilador **não resolve os endereços de símbolos externos**, ou seja variáveis globais e funções que estejam implementadas **em outros módulos ou em bibliotecas** (inclusive a biblioteca padrão).

O **linker**, resolve esses endereços.

Por exemplo:

```
call printf
```

será substituído por

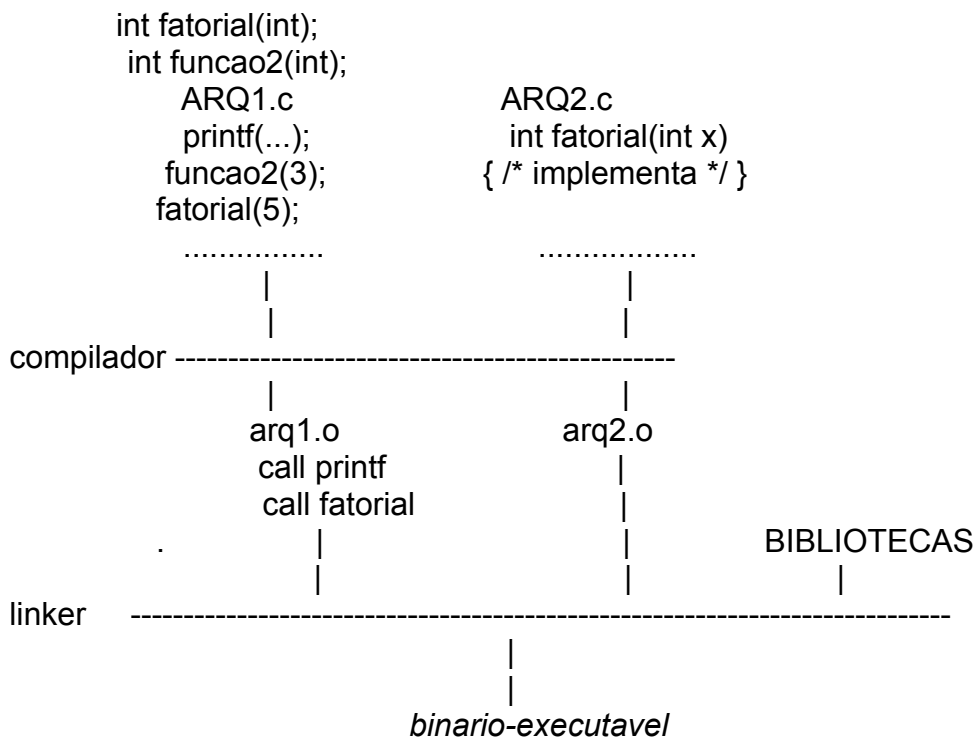
```
call <endereço-de-printf>
```

```
#include <stdio.h>
```

// os includes só são utilizados pelo compilador

// contem:

- regras de sintaxe (como structs, protótipos, etc),
- funções inline
- templates





---

## Anexo B. Respostas às questões para revisão

---

1. Questões do Capítulo 1.....	425
2. Questões do Capítulo 2.....	427
1. Exercício.....	427
2. Questões.....	428
3. Questões do Capítulo 3.....	431
1. Exercício.....	431
2. Questões.....	433
4. Questões do Capítulo 4.....	436
1. Exercício.....	436
2. Questões.....	440
5. Questões do Capítulo 5.....	446
1. Exercício 1.....	446
2. Exercício 2.....	452
3. Questões.....	455
6. Questões do Capítulo 6.....	468
7. Questões do Capítulo 7.....	471
8. Questões do Capítulo 11.....	473



---

## 1. Questões do Capítulo 1

---

Estas questões foram propostas na seção 1.11, página 38.

Cap.1 - 1. Com relação à padronização das linguagens **C** e **C++**:

- a. ☒ **C** é uma linguagem padronizada.
- b. ☒ **C++** é uma linguagem padronizada.
- c. ☐ **C não é uma linguagem padronizada.**  
{  
    **Errado.** A linguagem **C** é padronizada (C89 e C99)  
}
- d. ☐ **C++ não é uma linguagem padronizada.**  
{  
    **Errado.** A linguagem **C++** é padronizada (C++98, C++03).  
}
- e. ☐ *A linguagem **C** já foi completamente padronizada, mas a padronização de **C++** ainda está em sua fase inicial.*  
{  
    **Errado.** Já existe um padrão oficialmente estabelecido; o que não impede que novos padrões sejam criados, como uma evolução, mantendo compatibilidade com o padrão existente.  
}
- f. ☐ *O padrão da linguagem **C++** foi publicado em 1998, com uma atualização mínima em 2003. Isso é definitivo. Nunca mais haverá um novo padrão de **C++**.*  
{  
    **Errado.** Como afirmado na apostila, um novo padrão para **C++** está atualmente em fase de finalização: o **C++0x**.  
}

Cap.1 - 2. Assinale as afirmações verdadeiras sobre **C++**:

- a. ☐ **C++** introduziu apenas **pequenas** melhorias com relação ao **C**.  
{  
    **Errado.** Existem realmente melhorias que podemos chamar de “pequenas”. Mas as principais melhorias de **C++** com relação a **C**, são importantes a ponto de propiciar **novas técnicas de programação**, suportadas diretamente (ou nativamente) pela linguagem.  
}
- b. ☐ **C++** é baseada **apenas** em **C**.  
{  
    **Errado.** As linguagens **Simula**, principalmente, mas também **ADA**, **ALGOL** e **CLU**, em maior ou menor escala, foram referências importantes para a criação de **C++**, o que explica sua amplitude (múltiplas técnicas de programação). Certamente, a linguagem **C** é uma importantíssima referência para **C++**, mas essa influência **não é exclusiva**.  
}

- 
- }
- 
- c. ☒ **C++** é baseada em **C**, mas também herdou recursos decisivos de outras linguagens, o que torna **C++** muito diferente de **C**, em termos de técnicas de programação suportadas diretamente (ou nativamente).
- 
- d. ☐ *Além do **C**, a outra linguagem que também serviu de inspiração para a criação do **C++** foi **JAVA**.*
- {
- Errado.** As linguagens **C++** e **SmallTalk** herdaram da linguagem **Simula** a noção de orientação a objetos.
- C++** é também herdeira de outras linguagens, mas **não de Java**.
- Ao contrário**, é **Java** que descende de **C++** (e de **SmallTalk**).
- }
- 
- e. ☐ ***C++** suporta **apenas** uma **única** técnica de programação conhecida como "orientação a objetos"*
- {
- Errado.** **C++** admite **múltiplas** técnicas de programação
- }
- 
- f. ☒ **C++** é uma linguagem multi-paradigma, admitindo **múltiplas** técnicas de programação.
- 

**Cap.1 - 3.** Há uma série de áreas **dependentes de plataforma** não cobertas pela biblioteca padrão de **C++**. Sendo assim, o programador precisará:

- 
- a. ☐ *Para resolver esse problema, o programador deve desenvolver **sempre** suas próprias bibliotecas, criando o código de infra-estrutura que cuide dos detalhes dessas plataformas, e sirva de base para a criação de aplicações que possam ser compiladas em qualquer uma delas.*
- {
- Errado.** Existem bibliotecas disponíveis, *open source*, suportadas por comunidades amplas de programadores e/ou empresas. Além disso, há também bibliotecas comerciais para certas áreas específicas.
- Para que reinventar a pólvora? **Produtividade é essencial.**
- Bibliotecas próprias só se justificam para resolver problemas específicos de um projeto, ou para áreas nas quais não encontremos nada que já esteja pronto.
- }
- 
- b. ☒ Já existem boas bibliotecas que resolvem a maior parte desses problemas, como, por exemplo, **boost** e **Qt**. Bastará escolher e usar a(s) biblioteca(s) mais adequada(s) para resolver esse tipo de problema, podendo assim criar aplicações que rodam em diversas plataformas.
- 
- c. ☐ *Os **compiladores** são responsáveis por resolver esse tipo de problema.*
- {
- Errado.** Compiladores servem apenas para **analisar a sintaxe** utilizada pelo programador e, se estiver correta, **traduzir o código fonte para código de máquina**.
- Compiladores não criam código original.
- }
-

- d. ☐ Os **ambientes de desenvolvimento** são responsáveis por resolver esse tipo de problema.

{

**Errado.** Ambientes integrados de desenvolvimento (IDE's) servem para elevar a produtividade do programador, **integrando** editores de texto, chamadas ao compilador, chamadas ao programa de detecção de erros (*debugger*), podendo também permitir acesso rápido à documentação, ou outras atividades auxiliares.

Há ambientes de desenvolvimento que **geram código**. Mas, nesse caso, **eles se baseiam em alguma(s) biblioteca(s) já existente(s) - e dela(s) fazem uso** em situações muito comuns e previsíveis, que não exijam um desenvolvimento lógico específico.

}

## 2. Questões do Capítulo 2

### 1. Exercício

Este exercício foi proposto na seção 2.7.1, página 68.

**Enunciado:** com base no exemplo da função "Maximo", escreva a função "Minimo", no arquivo /cursoCPP/00\_funcoes/main.cpp (acima da função "main").

**Resultado que deve ser impresso (ou semelhante):**



```
C:\ Prompt de comando
0 maior valor entre 10 e 20 = 20
0 menor valor entre 10 e 20 = 10
```

**Passos para atingir o objetivo:**

- a. A função "Minimo" deve:
  - **Selecionar** o **menor** de dois valores inteiros e **retornar** esse valor.
- b. Em seguida, na função "main":
  - **Chame** a função "Minimo", passando-lhe os valores apropriados.
  - **Imprima** o resultado (retorno) devolvido pela função "Minimo".

**Solução (obs.: uma parte do código já foi escrita; considere os trechos assinalados como acréscimos novos):**

```
#include <iostream>
int Maximo( int x , int y )
{
    if ( x > y ) // se "x" é maior-que "y"
        return x ; // retorna um resultado
    else // do contrário
        return y ; // retorna outro resultado
}
```

// acrescentei : U

```
int Minimo( int x , int y )
{
```

```

if ( x < y ) // se "x" é menor-que "y"
    return x ; // retorna um resultado
else // do contrário
    return y ; // retorna outro resultado
}

```

// o programa **inicia aqui**:

```

int main()
{
    int a, b, c ;
    a = 10 ;
    b = 20 ;
    c = Maximo ( a, b ) ;
    std::cout << "O maior valor entre " << a << " e " << b
                << " = " << c << "\n" ;

```

// acrescentei : ↻

```

    c = Minimo ( a, b ) ;
    std::cout << "O menor valor entre " << a << " e " << b
                << " = " << c << "\n" ;

```

return 0 ;

}

c. **Compilar**, em um ambiente ou chamando na linha de comando:

```

cl /EHsc /Fe"00_funcoes" main.cpp // compilador microsoft - Windows
g++ main.cpp -o 00_funcoes // compilador gcc - Windows ou Unix/Linux

```

d. **Executar**, em um ambiente ou chamando na linha de comando:

```

00_funcoes (Windows)
./00_funcoes (Unix/Linux)

```

## 2. Questões

Estas questões foram propostas na seção **2.7.2**, página **69**.

Cap.2 - 1. A respeito de **funções**, assinale as afirmações corretas:

- a. ☐ Uma **função** é o mesmo que uma linha de instrução.
- {  
    **Errado.** Uma função contém um bloco de instruções (ainda que esse bloco seja constituído por uma única instrução).  
}
- b. ☐ Uma função é um bloco contendo um conjunto de linhas de instrução **podendo ou não** ter um nome.
- {  
    **Errado.** Uma função realmente é um bloco (um conjunto) de instruções, mas esse bloco não pode ser anônimo. A função sempre deve ter um nome.  
}
- c. ☒ Uma função é um bloco contendo um conjunto de linhas de instrução, e obrigatoriamente deve ter um **nome** que a identifique.
- d. ☒ Nomes de função são seguidos **obrigatoriamente** por parênteses.
- e. ☐ O bloco de instruções de uma função é **iniciado** pelo próprio nome da

*função e encerrado com a instrução return.*

```
{
    Errado. Uma função é iniciada e encerrada por chaves de abertura e
    fechamento: { ... }
}
```

- f. ☒ O bloco de instruções de uma função é **iniciado e encerrado** com chaves: { ... }
- g. ☐ Se uma **função** tiver apenas uma **única linha de instrução**, as chaves podem ser **omitidas**.
- ```
{
    Errado. Em uma função, nunca podemos omitir as chaves de abertura e encerramento.
}
```
- h. ☒ A expressão [ **c = Maximo ( a , b ) ;** ] contém uma **chamada** de função.

**Cap.2 - 2.** Sobre **linhas de instrução**, assinale as afirmações corretas:

- a. ☐ Uma linha de instrução é encerrada pela quebra de linha do editor de textos.
- ```
{
    Errado. Uma linha de instrução é encerrada com um ponto e vírgula.
}
```
- b. ☒ Uma linha de instrução é encerrada com um ponto e vírgula.

**Cap.2 - 3.** Considerando o código abaixo, assinale as afirmações corretas:

```
int x , y ;
// ...
if ( x > y )
std::cout << "x é maior que y" << "\n";
std::cout << "agora vou encerrar\n" ;
```

- a. ☐ **Apenas se "x" for maior que "y", será impresso x é maior que y e agora vou encerrar.**
- ```
{
    Errado. Se "x" for maior que "y" será impresso x é maior que y. Em seguida, sempre será impresso agora vou encerrar. Pois, como as chaves foram omitidas, a única instrução a executar caso a condição seja verdadeira é a linha do primeiro "cout".
    Para que essa afirmação fosse verdadeira, seria necessário que o código estivesse escrito assim:
```

```
if ( x > y )
{
    std::cout << "x é maior que y" << "\n";
    std::cout << "agora vou encerrar\n" ;
}
```

```
}
```

- 
- b. ☐ *Nada será impresso.*  
 {  
   **Errado.** Pelo menos o segundo "cout" será executado e assim **sempre** será impresso **agora vou encerrar**.  
 }
- 
- c. ☒ **Apenas** se "x" for maior que "y", será impresso **x é maior que y**.  
 Em qualquer caso, **sempre** será impresso **agora vou encerrar**.
- 
- d. ☐ **Sempre** será impresso **x é maior que y e agora vou encerrar**.  
 {  
   **Errado.** Sempre será impresso **agora vou encerrar**. Mas **x é maior que y** só será impresso se o valor de 'x' for maior que o valor de 'y'.  
 }
- 

**Cap.2 - 4.** Uma aplicação tem seu **início** (ou ponto de entrada) em:

- 
- a. ☐ *Nesta linha: [ #include <iostream> ].*  
 {  
   **Errado.** Uma diretiva [ #include ] indica ao compilador que ele deve procurar, no arquivo especificado, pela declaração de símbolos usados em um arquivo de código fonte mas que não são declarados aí. Por exemplo, **cout** está declarado no arquivo **iostream**, e é lá que o compilador ficará sabendo como ele pode ser usado. Desse modo, isso **não tem nada a ver** com o **início** de uma aplicação.  
 }
- 
- b. ☐ *Na primeira função que esteja escrita em um **arquivo** que deve ter o nome de **main.cpp**.*  
 {  
   **Errado.** Não existe nenhuma convenção sobre nomes de arquivos especiais. O início de uma aplicação ocorre, independentemente de arquivo, na função **main**. Esta sim é uma convenção da linguagem, e por isso esse nome de função é especial: é aí que a aplicação tem início.  
 }
- 
- c. ☒ Em uma **função** que deve ter o nome especial de **main**, não importando o nome do arquivo em que esteja escrita. Essa função é obrigatória em qualquer aplicação executável.
- 

**Cap.2 - 5.** A respeito do **compilador** podemos afirmar que:

- 
- a. ☐ *O compilador serve apenas para analisar se o código foi escrito corretamente segundo as regras da linguagem.*  
 {  
   **Errado.** Além de analisar a correção da escrita (sintaxe e gramática da linguagem), um compilador deve também traduzir o código fonte para linguagem de máquina, caso esteja tudo correto.  
 }
- 
- b. ☒ Ele analisa se o código foi escrito corretamente, de acordo com a linguagem, e, em caso positivo, irá traduzi-lo para linguagem de máquina.
- 
- c. ☐ *O compilador indica erros de escrita, mas não fornece informações so-*
-

*bre esse erro.*

```
{  
    Errado. É responsabilidade do compilador fornecer uma descrição  
    sobre o tipo de erro, e indicar a linha onde o mesmo ocorreu.  
}
```

- d. ☒ O compilador indica erros de escrita, emitindo uma mensagem sobre o tipo do erro e a linha em que ele ocorreu. **Devemos usar essas duas informações** para corrigir os erros mais rapidamente.



**Para fazer:** crie um **projeto de teste** para as questões deste capítulo (quando isso **seja possível**). Procure **reproduzir, no código**, as situações descritas nas questões utilizadas. Introduza algumas **variações**.

### 3. Questões do Capítulo 3

#### 1. Exercício

Este exercício foi proposto na seção **3.6.1**, página **91**.

**Enunciado:** implemente a soma dos números entre um número **inicial** e um **final**, com a distância "**razão**" entre cada número. Esses 3 valores devem ser informados. E o usuário deverá decidir também se deseja um novo cálculo ou encerrar o programa.

**Resultado que deve ser impresso (ou semelhante):**

```
C:\> Prompt de comando  
informe os numeros inicial, final e a razao  
- nessa ordem:  
10  
20  
2  
resultado da soma: 90  
  
deseja realizar novo calculo?  
<0 para encerrar,  
qualquer outro numero para continuar>  
0  
  
fim de processamento
```

**Modelo  
esquemático:**

```

while ( < ...> )
{
    // a) entrada de dados
    if ( <erro_na_entrada> )
        // imprime mensagem de erro
    else
    {
        for ( < ...> ; < ...> ; <...> )
        {
            // b) soma...
        }
        // c) imprime resultado
        // d) pergunta ao usuário: deseja continuar?
        // e) a entrada do usuário pode modificar a condição
        // do laço "while", interrompendo o laço.
    } // fim do if
} // fim do while

```

- Lembrando: para a soma, usar o **laço for** e não o cálculo de soma dos termos da progressão, pois queremos exercitar o laço.

**Solução:**

```

#include <iostream>
#include <limits>
int main()
{
    int continuar = 1; // flag para continuidade do programa
    while ( continuar ) // enquanto for verdadeiro...
    {
        // solicita ao usuário os 3 números necessários:
        int inicial, final, razao, result ;
        std::cout << "\ninforme os números inicial, final e a razão\n" ;
        std::cin >> inicial >> final >> razao ;
        // se houve falha na entrada:
        if ( std::cin.fail( ) )
        {
            std::cout << "valores incorretos: tente outra vez\n";
            // como houve falha é preciso limpar os flags de erro de "cin":
            std::cin.clear( );
            // e desprezar quebras de linha pendentes no buffer de "cin":
            std::cin.ignore(std::numeric_limits<int>::max(), '\n');
        }
        else
        {
            if ( razao == 0 ) // analisa "razao"
                std::cout << "razao não pode ser zero - tente outra vez\n";
            else
            {
                if ( inicial > final ) // se "inicial" é maior-que "final":
                    std::swap( inicial, final ); // troca valores de "inicial" e "final";
                // agora "inicial" é o menor
            }
        }
    }
}

```



```

if ( razao < 0 ) // neste caso, "razao" deve ser positivo,
{
    // para evoluir do menor para o maior
    razao = -razao; // "razao" passa de negativo para positivo
    std::cout << "razao alterada para : " << razao << "\n";
}

// o laço para cálculo da soma:
for ( result = 0 ; inicial <= final ; inicial=inicial+razao )
    result = result + inicial ; // executa se condição verdadeira

std::cout << "resultado da soma: " << result << "\n";

std::cout << "\ndeseja realizar novo calculo?\n"
    << " (0 para encerrar, qualquer outro numero "
    << "para continuar)\n";

std::cin >> continuar ; // o usuário pode ter digitado zero...

```



A partir daqui, o valor de "**continuar**" poderá ser **verdadeiro** ou **falso** (zero), afetando a próxima avaliação de **condição** do laço "**while**".

```

if ( std::cin.fail() ) // deve ter digitado uma letra...
{
    // consideremos isso como falso
    std::cout << "entrada incorreta - " <<
        "entendida como encerramento\n" ;

    continuar = 0 ; // com isso, o laço "while" será encerrado
}

} // fim do "if (razao == 0 )"

} // fim do primeiro "if (std::cin.fail( ) )"

} // fim do laço "while"


std::cout << "\nfim de processamento\n";
return 0;

} // fim de "main".

```

## 2. Questões

Estas questões foram propostas na seção **3.6.2**, página **94**.

 Nas questões abaixo, considere as seguintes declarações iniciais:  
**int x , y ;**

**Cap.3 - 1.** A respeito de uso da **memória**, assinale as afirmações corretas:

- a. ☐ *Não é preciso ter preocupações especiais com a memória de um computador. Isso é resolvido pelo compilador.*
- Errado.** Devemos declarar memórias (constantes ou variáveis) com um determinado **tipo**. Isso permite evitar enganos (pois o compilador avisará sobre usos incorretos) e também elevar a *performance*, pois não será necessário, durante a execução, avaliar que tipo de memória está sendo alocada a cada momento: pois isso já foi resolvido na compilação.

- b. ☒ Uma memória deve ser reservada e qualificada com um determinado **tipo** pelo programador. Por exemplo, usando **int**, como na expressão `[ int x ; ]`
- c. ☐ A expressão `[ x = 10 ; ]` visa **confirmar** se '**x**' contém o valor **10**.  
 {  
     **Errado.** Essa expressão não contém uma comparação (ou confirmação). E sim uma **atribuição**: copie **10** para '**x**'.  
 }
- d. ☒ A expressão `[ x = 10 ; ]` visa **armazenar** o valor **10** em '**x**', copiando esse valor para a área de memória à qual associamos o nome '**x**'.

### Cap.3 - 2. Sobre **fluxo de processamento**, podemos dizer que:

- a. ☐ O processamento é **sempre** baseado na execução sequencial de um certo número de instruções.  
 {  
     **Errado.** A execução em princípio é sequencial, mas, nessa sequência, podemos ter testes de decisão que podem conduzir a diferentes caminhos de execução, dependendo de uma avaliação de condição.  
 }
- b. ☒ O processamento pode ser bifurcado em caminhos diferentes, através de **tomadas de decisão**.
- c. ☐ A expressão `[ if ( x > y ) ]` contém uma operação que leva à seguinte avaliação: "será que **x** pode ser **armazenado** em **y**" ?  
 {  
     **Errado.** Essa expressão contém uma comparação **maior-que**.  
 }
- d. ☒ Em `[ if ( x > y ) ]` temos uma operação que implica nesta avaliação: "o valor armazenado em '**x**' é **maior-que** o valor armazenado em **y**" ?
- e. ☒ Após `[ if ( x > y ) ]` o processamento poderá seguir um caminho diferente se o resultado da avaliação dessa condição for verdadeiro

### Cap.3 - 3. Sobre os conceitos de **verdadeiro** e **falso**, podemos dizer que:

- a. ☒ A avaliação da expressão contida em `[ if ( x > y ) ]` retornará um resultado **verdadeiro** se '**x**' for **maior-que** '**y**'; do contrário retornará um resultado **falso**.
- b. ☐ A avaliação da expressão contida em `[ if ( x >= y ) ]` retornará um resultado **verdadeiro** se '**x**' for **maior-que** '**y**'; do contrário retornará um resultado **falso**  
 {  
     **Errado.** O resultado será verdadeiro se '**x**' for **maior-ou-igual** a '**y**', pois o operador empregado foi `[ >= ]` e não `[ > ]`.  
 }
- c. ☐ A expressão contida em `[ if ( x ) ]` **não é aceita** pelas linguagens **C** e **C++**. Logo, ocorrerá um erro de compilação.  
 {

**Errado.** Um valor pode ser avaliado como verdadeiro ou falso. Se for **zero**, a avaliação retorna **falso**. Se **diferente de zero**, retornará **verdadeiro**.

}

- d. ☐ A avaliação da expressão contida em [ **if ( x )** ] retornará um resultado **falso** se o valor de '**x**' for igual a **0(zero)**; e retornará um resultado **verdadeiro** se o valor de '**x**' for **exatamente** igual a **1(um)**.

{

**Errado.** Ver a resposta a resposta '**e**', abaixo.

}

- e. ☒ A avaliação da expressão contida em [ **if ( x )** ] retornará um resultado **falso** se o valor de '**x**' for igual a **0(zero)**; e retornará um resultado **verdadeiro** se o valor de '**x**' for **diferente** de **0(zero)**

**Cap.3 - 4.** O que será impresso na execução do código abaixo?

```
x = 5 ; y = x + 1 ; x = x + 2 ;
if ( x >= y )
    std::cout << "x é maior ou igual a y\n" ;
else
    std::cout << "x é menor que y\n" ;
```

- a. ☒ Será impresso **x é maior ou igual a y**

- b. ☐ Será impresso **x é menor que y**

{

**Errado.** Só será impresso [ **x é menor que y** ], se [ **x >= y** ] for **falso**, ou seja se '**x**' for **menor-que** '**y**'. No caso, '**y**' contem **6** ( [ **y = 5 + 1** ] ) e '**x**' contem **7** ( [ **x = 5 + 2** ] ). Logo '**x**' é **maior que** '**y**'.

}

- c. ☐ **Nada** será impresso.

{

**Errado.** Seja lá qual for o resultado da avaliação da condição [ **x >= y** ], algo será impresso. Se verdadeira, temos um caminho de execução. E, se falsa, temos outro caminho de execução. Uma dessas duas impressões terá que ser executada.

}

- d. ☐ Será impresso **x é maior ou igual a y** e, **em seguida**, será impresso **x é menor que y**

{

**Errado.** Apenas uma dessas duas cadeias de caracteres será impressa: [ **if (...)** <executa-uma-coisa> ; **else** <executa-outra-coisa> ; ].

}

**Cap.3 - 5.** Sobre **laços**, podemos dizer que:

- a. ☐ Um laço **sempre** repete, **infinitamente**, um certo número de instruções.

{

**Errado.** Todo laço contem uma avaliação de condição. Dependendo dessa avaliação, o laço poderá prosseguir ou será interrompido. A repetição só será infinita se a avaliação retornar **sempre** um resultado **verdadeiro**.

- 
- }
- 
- b. ☒ Um laço pode conter uma avaliação de condição que, caso seja ou torne-se falsa, resultará na interrupção da repetição das instruções a ele associadas.
- 
- c. ☐ O laço **while** contém uma avaliação de condição que permite interrompê-lo. Já o laço **for** contém apenas uma possibilidade de **progressão**, mas não prevê uma avaliação de condição.
- {
- Errado.** O laço **for**, como qualquer laço, contém uma avaliação de condição: [ **for** ( <início> ; <condição> ; <progressão> ) ]
- }
- 

**Cap.3 - 6.** Quantas vezes serão executadas as instruções associadas ao seguinte laço:

**for** ( **x = 1** ; **x < 1** ; **x = x + 1** ) { /\* instruções \*/ }

---

- a. ☐ **Uma vez.**
- {
- Errado.** A instrução de início [ **x = 1** ; ] é executada na primeira vez e, **imediatamente em seguida**, é executada a **avaliação da condição**. As <instruções> só serão executadas se o resultado da avaliação for verdadeiro.
- E, no caso, o valor de '**x**' é **1** e a condição é [ **x < 1** ]. Logo, a avaliação tem resultado **falso**, pois '**x**' **não é menor-que 1**.
- Desse modo, a **primeira avaliação** já tem resultado **falso** e as <instruções> **nunca** serão executadas.
- }
- 
- b. ☐ **Dois vezes.**
- {
- Errado.** Ver resposta 'a', acima.
- }
- 
- c. ☒ **Nunca** serão executadas.
- 
- d. ☐ *Serão executadas infinitamente.*
- {
- Errado.** Ver resposta 'a', acima.
- }
- 



**Para fazer:** crie um **projeto de teste** para as questões deste capítulo (quando isso **seja possível**). Procure **reproduzir, no código**, as situações descritas nas questões utilizadas. Introduza algumas **variações**.

## 4. Questões do Capítulo 4

---

### 1. Exercício

Este exercício foi proposto na seção **4.11.1**, página **124**.

---

**Enunciado:**

---

- Crie um **vetor de inteiros** que deverá armazenar todos os números entre um número **inicial** e um **final**, existindo entre eles uma distância denominada "**razão**".
- O "inicial", o "final" e a "razão" devem ser **informados pelo usuário**.
- Após o armazenamento dos números no vetor, o programa deve **imprimir todos os elementos do vetor**.
- Esses números **também devem ser inseridos em uma string**, formando uma cadeia de caracteres. Por isso, o intervalo entre 'inicial' e 'final' deve ser **compatível** com o tipo **char** (por exemplo, entre 'A' e 'Z', ou outro).
- Ao final, **imprimir**: o **total de números** no intervalo, a **soma dos elementos** do vetor e a **string** formada pelos números.
- O programa deve também permitir que o usuário **repita a operação**, informando novos valores.

**Resultado que deve ser impresso:**

```

C:\ Prompt de comando
progressao com caracteres
Informe Inicial, Final e Razao.
Limites: A <= Inicial <= Final <= Z.
Razao: maior que zero
e menor ou igual a <Final-Inicial+1>.
A
Z
12
65
77
89
Total de numeros no intervalo: 3
Resultado da soma: 231
String: AMY
deseja realizar novo calculo?
<0 para encerrar, 1 para continuar>
0
fim de processamento
  
```

**Solução:**

```

#include <iostream>
#include <string>
#include <vector>
#include <limits>

int main()
{
    std::vector< int > vec;
    std::string str;
    std::cout << "progressao com caracteres\n";

    // limites para entrada de dados de 'inicial' e 'final'
    const char inicial_min = 'A';
    const char final_max = 'Z';

    bool continuar = true; // flag para continuidade do programa
    while ( continuar ) // enquanto for verdadeiro
    {
        // flag para indicar se os dados foram informados corretamente:
        bool valores_validos = true;

        // solicita ao usuário o inicial, o final e a razão:
        char inicial, final;
  
```

```

int razao ;
std::cout << "\nInforme Inicial, Final e Razao.\n"
            << "Limites: " << inicial_min
            << " <= Inicial <= Final <= " << final_max << ".\n"
            << "Razao: maior que zero \n"
            << "e menor ou igual a (Final-Inicial+1).\n" ;

std::cin >> inicial >> final >> razao ;

// Se houve falha na entrada:
if ( std::cin.fail( ) )
{
    std::cout << "valores incorretos\n";
    // como houve falha é preciso limpar os flags de erro de "cin":
    std::cin.clear( );
    // e desprezar quebras de linha pendentes no buffer de "cin":
    std::cin.ignore(std::numeric_limits<int>::max(), '\n');

    valores_validos = false; // flag: valores inválidos.
}

// Se não houve erro até aqui:
if ( valores_validos )
{
    // converte 'inicial' e 'final' para "uppercase" (caso não seja):
    inicial = char(toupper(inicial)); // se 'a' -> 'A'...
    final   = char(toupper(final));   // se 'b' -> 'B'...
}

```

A função "**toupper**" da biblioteca padrão retorna "**int**". Mas a conversão para "**char**" será segura já que abaixo checaremos os limites de "**inicial**" e "**final**".

```

// Analisa todas as regras e emite todas as mensagens cabíveis:
if ( razao <= 0 )
{
    std::cout << "Razao deve ser maior que zero.\n";
    valores_validos = false;
}

if ( inicial < inicial_min )
{
    std::cout << "Inicial deve ser maior ou igual a "
              << inicial_min << ".\n";
    valores_validos = false;
}

if ( final > final_max )
{
    std::cout << "Final deve ser menor ou igual a "
              << final_max << ".\n";
    valores_validos = false;
}

if ( inicial > final )
{
    std::cout << "Inicial deve ser menor ou igual a Final.\n";
    valores_validos = false;
}

if ( razao > (final-inicial+1) )
{
    std::cout << "Razao nao pode ser maior que "
              << " (Final-Inicial+1).\n";
    valores_validos = false;
}
}

```

```
// se tudo ocorreu corretamente:
if ( valores_validos )
{
    // Quantidade total de números no intervalo:
    unsigned int quant_num = (final-inicial+razao) / razao ;
```



OBS: a **conversão** do resultado das operações aritméticas acima, o qual terá o tipo **"int"** (o tipo de 'razao') para a variável **"quant\_num"**, que é **"unsigned int"**, seria insegura caso pudesse ter um valor **negativo**. Além disso o resultado **não** pode ser **zero**, pois servirá para **dimensionar o vetor e a string**. Mas neste caso nada disso acontecerá (e teremos uma conversão segura) pois:

- garantimos que 'inicial' e 'final' estarão dentro de um intervalo positivo e que 'inicial' não será maior que 'final'.
- garantimos que 'razao' não pode ser maior que [ inicial - final + 1 ] e desse modo o resultado da operação não será zero.
- finalmente, garantimos também que 'razão' não pode ser zero (impedindo a divisão por zero).

Agora podemos usar, coerente e seguramente, "quant\_num" como dimensão do vetor e da string:

```
vec.resize( quant_num ); // Dimensiona o vetor (garantimos que
                          // dimensão é maior que zero).
str.resize( quant_num ); // Dimensiona a string (idem).

unsigned int index;

char r = char(razao);
```



Esta conversão visa evitar que o último segmento do laço **"for"** abaixo [ **inicial = inicial + razao** ] receba uma eventual "warning" do compilador. Pois como "razao" é **int**, poderia conter um valor que, somado a 'inicial', não coubesse na atribuição que é feita a 'inicial' [ **inicial = <int>** ].

Acima, garantimos que isso não poderá acontecer, mas o compilador não é obrigado a analisar toda a lógica do programa. Ele apenas verifica uma possível truncagem de um determinado valor. Assim, ao invés de [ **inicial = inicial + razao** ], fazemos [ **inicial = inicial + r** ], com **todos os tipos compatíveis**:

```
for ( index = 0 ; inicial <= final ; inicial = inicial + r )
{
    vec[ index ] = inicial; // Conversão de "char" para "int": segura.
    str[ index ] = inicial; // Alimenta a string (OK: 'inicial' é "char").
    index = index + 1;      // Evoluir o índice.
}

// Acumular os elementos do vetor e imprimi-los:
int soma = 0;
for ( index = 0; index < vec.size(); index = index + 1 )
{
    soma = soma + vec[ index ];
    std::cout << vec[ index ] << "\n";
}

std::cout << "\nTotal de numeros no intervalo: "
          << quant_num << "\n";
std::cout << "Resultado da soma: " << soma << "\n";
std::cout << "String: " << str << "\n";

std::cout << "\ndeseja realizar novo calculo?\n"
          << "(0 para encerrar, 1 para continuar)\n\n";
```

```
std::cin >> continuar ;
```



A partir daqui, o valor de "continuar" poderá ser **verdadeiro ou falso** afetando **próxima avaliação de condição** do laço "while"

```
if ( std::cin.fail() ) // Deve ter qualquer coisa diferente de zero ou um...
                        // consideremos isso como falso:
{
    std::cout << "entrada incorreta - entendida como "
                << "encerramento\n" ;
    continuar = false ; // o laço "while" será encerrado
}
std::cout << "\n";

} // --- fim do último "if (valores_validos)"

else // Um (ou mais) valor inválido:
    std::cout << "Tente outra vez\n";

} // -- fim do laço "while"

std::cout << "fim de processamento\n";
return 0;

} // - fim de main.
```

## 2. Questões

Estas questões foram propostas na seção 4.11.2, página 128.

Cap.4 - 1. Assinale as respostas corretas, considerando estas declarações:

```
long i ;
float f ;
```

- a. ☐ 'i'(long) tem o mesmo **tamanho** de 'f'(float): 4 bytes. Logo podemos dizer que não há qualquer diferença quanto ao modo como essas variáveis serão representadas na memória.
- {  
**Errado.** Embora tenham o mesmo tamanho, esses dois tipos (**long** e **float**) têm forma de armazenamento diferentes: **long** é armazenado como um número **inteiro** e **float** é armazenado através de **ponto flutuante**.  
}
- b. ☐ 'i'(long) tem uma **forma de armazenamento** diferente de 'f'(float), já que esta é representada na memória através de ponto flutuante. Logo, apesar de terem o mesmo tamanho, são representadas de modo **diferente**. Mas isso não tem **nenhuma** implicação prática.
- {  
**Errado.** A forma de armazenamento do **float** (ponto flutuante) é mais complexa, implicando em **acesso mais lento**, do que a forma de armazenamento do **long** (número **inteiro**).  
}
- c. ☒ A forma de armazenamento de 'f'(float), através de ponto flutuante, torna o seu acesso mais lento do que o acesso a números inteiros.



**Cap.4 - 2.** Assinale as respostas corretas, considerando o código abaixo:

```
const int constante ;  
//...  
constante = 5 ;
```

- a. ☐ As duas linhas do código acima estão **corretas**.  
{  
    **Errado.** Uma constante, por sua própria natureza, deve ser obrigatoriamente **inicializada**, o que **não foi feito**, e, além disso **não pode sofrer atribuições** (ou não seria constante), e isso **foi feito**.  
}
- b. ☐ A primeira linha do código acima está **incorreta**, pois uma declaração **const** exige **inicialização**. A segunda linha está **correta**.  
{  
    **Errado.** A primeira afirmação está **correta** (exige **inicialização**). Mas a segunda afirmação está **incorreta** ao dizer que a segunda linha [ **constante = 5 ;** ] está correta. Temos aí uma **atribuição** e constantes não podem receber atribuições.  
}
- c. ☒ A primeira linha do código acima está **incorreta**, pois uma declaração **const** exige **inicialização**. A segunda linha também está **incorreta**, pois constantes podem apenas ser inicializadas mas não podem sofrer **atribuições**.

**Cap.4 - 3.** Assinale as respostas corretas, considerando o código abaixo:

```
int variavel ;  
//...  
variavel = 5 ;
```

- a. ☐ As duas linhas do código acima estão **incorretas**.  
{  
    **Errado.** A primeira linha está correta: declara uma variável do tipo **int** - que não é inicializada. Nada de anormal, pois variáveis podem ou não ser inicializadas (**a inicialização não é obrigatória**). A segunda linha também está correta pois **variáveis** podem, a qualquer momento, receber **atribuições**.  
}
- b. ☐ A primeira linha do código acima está **incorreta**, pois uma declaração de variável exige **inicialização** obrigatoriamente. A segunda linha está **correta**.  
{  
    **Errado.** Para variáveis, a inicialização é **opcional**. Ver resposta 'a', acima.  
}
- c. ☒ As duas linhas do código acima estão **corretas**.

**Cap.4 - 4.** Assinale as respostas corretas, considerando estas declarações:

```
double d = 10.9 ;  
int i = d ;
```

- a. ☐ O código acima não apresenta **nenhum tipo de problema**.  
 {  
     **Errado.** Dependendo da situação real, isso poderá ser um problema. Ver resposta 'b', abaixo.  
 }
- b. ☒ A variável 'i'(int) não tem suporte a ponto flutuante. Logo o valor de 'd' (10.9) será truncado para 10 ao ser copiado para 'i'. Isso pode ser um problema, dependendo da situação em que ocorra. E, se isso está sendo feito de modo **intencional**, deveria ser usada uma operação de **cast** (conversão) **explícita**, pois, em princípio, essa é uma operação **suspeita**.

Cap.4 - 5. Assinale as respostas corretas, considerando estas declarações:

```
double d = double (4) * 1024 * 1024 * 1024 ;
long i = d ;
```

```
// OBS.: o resultado de [ double (4) * 1024 * 1024 * 1024 ]
//      é 4.294.967.296.
```

- a. ☐ O código acima não apresenta **nenhum tipo de problema**.  
 {  
     **Errado.** Embora o resultado das operações de multiplicação, que é atribuído a 'd', tenha sua **parte fracionária zerada** (e assim, neste aspecto, nada será truncado), na **parte inteira** temos um valor muito grande para caber em um **long**: **4.294.967.296**. Então, na atribuição a 'i' (*long*), a parte inteira será truncada para – **2.147.483.648**. Dependendo da situação, isso pode ser um problema (perda de valor). E se isso foi usado de modo **intencional**, temos um **código confuso**. No mínimo deveria ser usada uma operação de **cast** (conversão) **explícita**, pois é uma operação **muito suspeita**.  
 }
- b. ☒ O valor de 'd' (*double*) será truncado ao ser copiado para 'i' (*long*)

Cap.4 - 6. Assinale as respostas corretas, considerando o código abaixo:

```
std::string nome ;
nome = "Maria" ;
nome = nome + " Aparecida" ;
```

- a. ☒ As três linhas do código acima estão **corretas** e não implicam em **nenhum tipo de problema**.
- b. ☐ As três linhas do código acima estão **incorretas**.  
 {  
     **Errado.** Nada de incorreto. "nome" foi declarada corretamente com o tipo **std::string**. Em seguida, houve uma operação de **atribuição**, totalmente suportada e garantida por esse tipo. Finalmente, uma operação de **concatenação**, igualmente suportada e garantida. Ver ainda a resposta 'c', abaixo.  
 }
- c. ☐ A terceira linha do código acima pode implicar em um problema, escrevendo em **memória ainda não alocada**.

---

```
{
    Errado. O tipo std::string garante que serão feitas as alocações de
    memória necessárias em suas operações. Assim, tanto na atribuição [
    ≡ ] como na concatenação [ += ], se houver necessidade de alocar
    mais memória, std::string cuidará disso.
}
```

---

**Cap.4 - 7.** Assinale as respostas corretas, considerando o código abaixo:

```
std::vector<int> vetor ;
vetor [ 0 ] = 10 ;
```

---

- a. ☐ As duas linhas do código acima estão **corretas** e não implicam em **nenhum tipo de problema**.

```
{
    Errado. As duas linhas estão sintaticamente corretas. Mas a segunda
    linha apresenta um problema potencial em tempo de execução. Ver
    resposta 'c', abaixo.
}
```

---

- b. ☐ As duas linhas do código acima estão **incorretas**.

```
{
    Errado. Não há nada de incorreto nessas duas linhas. Apenas, a
    segunda linha pode levar a erros de execução. Ver resposta 'c', abai-
    xo.
}
```

---

- c. ☒ A primeira linha do código acima está **correta**. A segunda linha apresen-  
ta um **problema potencial**, já que "**vetor**" não foi dimensionada e há  
um acesso ao seu primeiro elemento (**[0]**). Antes disso, alguma coisa  
deveria ter sido feita, como, por exemplo:  
**[ vetor.resize( <dimensão> ) ; ]**.  
Desse modo a execução da segunda linha implica em resultados **impre-  
visíveis** (ou indetermináveis).
- 

**Cap.4 - 8.** Assinale as respostas corretas, considerando o código abaixo:

```
enum Meses { Janeiro=1, Fevereiro, Marco, Abril, Maio, Junho, Julho,
             Agosto, Setembro, Outubro, Novembro, Dezembro } ;

std::vector< std::string > vec ;
vec.resize( Dezembro ) ;
vec[ Janeiro ] = "Janeiro" ;
// faz o mesmo para Fevereiro, Marco... até vec[Dezembro]="Dezembro";
unsigned int index;
for ( index = 0 ; index < vec.size() ; index = index + 1 )
    std::cout << vec[ index] << "\n" ;
```

---

- a. ☐ Todas as linhas do código acima estão **corretas** e não implicam em **nenhum tipo de problema**.

```
{
    Errado. Todas as linhas de código estão formalmente corretas, em
    conformidade com as regras da linguagem. Mas teremos um proble-  
ma potencial quando a linha [ vec [ Dezembro ] = "Dezembro" ; ]
    for executada, pois o vetor foi dimensionado para 12 elementos, e
```

---

portanto só pode ser acessado com os índices de **0** a **11**. A constante **Dezembro** do tipo **Meses** será convertida para **12** do tipo **int**.

Logo, `[ vec [ 12 ] = "Dezembro"; ]` está tentando acessar um décimo-terceiro elemento, para o qual não foi alocada memória. O correto seria fazer:

```
vec [ Janeiro -1 ] = "Janeiro"; // índice zero.
```

```
// ...
```

```
vec[ Dezembro -1 ] = "Dezembro"; // índice 11.
```

Ver, abaixo, a resposta à opção 'k'.

```
}
```

- b. ☐ *Todas as linhas do código acima estão **incorretas**.*

```
{
```

**Errado.** Todas as linhas do código acima estão formalmente corretas. Apenas uma delas, já apontada na resposta anterior, apresenta um problema potencial.

```
}
```

- c. ☐ *Não é possível fazer: `[ std::vector < std::string > vec ; ]`.*

*Só são possíveis os usos de `std::vector` para tipos básicos como:*

*`[ std::vector < int > vi ; ]` ou `[ std::vector < double > vd ; ]`.*

```
{
```

**Errado.** `std::vector<>`, permite criar um vetor de qualquer tipo que aceite a operação de atribuição, inclusive `std::string`.

```
}
```

- d. ☐ *Não é possível fazer: `[ vec.resize( Dezembro ) ; ]`.*

```
{
```

**Errado.** A constante **Dezembro** do tipo **Meses** pode ser convertida para **int**. Logo, nesse caso, temos o mesmo que `[ vec.resize(12); ]`.

```
}
```

- e. ☐ *Não é possível fazer: `[ vec [ Janeiro ] = "Janeiro" ; ]`.*

```
{
```

**Errado.** A constante **Janeiro** do tipo **Meses** pode ser convertida para **int**. Ou seja: nesse caso, temos o mesmo que:

```
[ vec[ 1 ] = "Janeiro"; ]
```

Logo, é **possível**, embora em função da situação lógica do exemplo, o correto seria fazer:

```
vec [ Janeiro -1 ] = "Janeiro"; // índice zero.
```

```
}
```

- f. ☒ A expressão `[ vec.resize( Dezembro ) ; ]` está correta pois "**Meses**" pode ser convertido para **int**. Nesse caso, seria o mesmo que: `[ vec.resize( 12 ) ; ]`.

- g. ☒ A expressão `[ vec [ Janeiro ] = "Janeiro" ; ]` está correta pois "**Meses**" pode ser convertido para **int**. Nesse caso, seria o mesmo que: `[ vec [ 1 ] = "Janeiro" ; ]`.

- h. ☐ *Caso não ocorra um erro de execução, o laço **for** do código acima irá imprimir:*

```
Janeiro
```

```
Fevereiro
```

```
// Marco, Abril ... Novembro
```

```
Dezembro.
```

```
{
  Errado. Como a atribuição de valores aos elementos do vetor começa com [ vec[ Janeiro ] = "Janeiro"; ] (que é o mesmo que [ vec[ 1 ] = "Janeiro"; ] ), o primeiro elemento do vetor (vec[ 0 ]), contém uma string vazia. Logo, ao imprimir o primeiro elemento, será impressa essa string vazia. Ver, acima, a resposta à opção 'a'.
  Além disso, não irá atingir Dezembro. Ver, abaixo, as resposta às opções 'i' e 'j'.
}
```

- i. ☐ *Caso não ocorra um erro de execução, o laço **for** do código acima irá imprimir:*

```
// <uma string vazia>
Janeiro
Fevereiro
// Marco, Abril ... Novembro
Dezembro.
```

```
{
  Errado. Realmente, a primeira coisa a ser impressa será uma string vazia. Mas a impressão não atingirá "Dezembro", pois o laço for está percorrendo o vetor para imprimir todos os elementos entre os índices 0 e [ vec.size() - 1 ] ( [ index < vec.size() ] ), ou seja de 0 a 11. Logo, a última string a ser impressa será Novembro.
}
```

- j. ☒ *Caso não ocorra um erro de execução, o laço **for** do código acima irá imprimir:*

```
// <uma string vazia>
Janeiro
Fevereiro
// Marco, Abril ... Outubro
Novembro.
```

- k. ☒ *É possível que ocorra um erro de execução na linha:*  
 [ **vec [ Dezembro ] = "Dezembro" ;** ], ou em algum momento qualquer após sua execução.  
 Isso porque o vetor foi **dimensionado** para **12** elementos e deve ser acessado com os índices de **0** a **11**.  
 Mas, nessa linha, está sendo acessado com o **índice 12** - pois esse será o resultado da conversão da constante **Dezembro**, do tipo **Meses**, para **int**. Logo, está tentando acessar um **décimo-terceiro** elemento, para o qual não há memória alocada.  
 Caso isso atinja uma área de memória não reservada para a aplicação em um sistema multi-tarefa, o SO irá interromper a aplicação.  
 Do contrário, teremos resultados indeterminados em operações seguintes, pois é possível que essa operação esteja escrevendo em memória alocada para outra finalidade.



**Para fazer:** crie um **projeto de teste** para as questões deste capítulo (quando isso **seja possível**). Procure **reproduzir, no código**, as situações descritas nas questões utilizadas. Introduza algumas **variações**.

## 5. Questões do Capítulo 5

### 1. Exercício 1

Este exercício foi proposto na seção 5.11.2, página 175.

**Enunciado:** criar um projeto com dois módulos e testar várias funções.

- Neste exercício serão criados **duas unidades de tradução** (ou módulos), isto é, **dois arquivos fontes**. O arquivo **main.cpp** deve conter **apenas a função main**. O arquivo **funcoes.cpp** deverá conter **diversas funções** que realizarão tarefas específicas. **main** deverá **testar todas as funções** implementadas no módulo **funcoes.cpp**. As funções que devem ser escritas no módulo **funcoes.cpp** são:
  - **unsigned long long Fatorial ( unsigned int num ) ;**  
**unsigned long long Potencia ( unsigned int base, unsigned int exp ) ;**  
**int PA\_TotalTermos ( int inicial, int final, int razao ) ;**  
**void ImprimePares ( int inicial , int final ) ;**  
**double DobraValor ( int ultimo\_dia ) ;**  
**double DobraValor\_for ( int ultimo\_dia ) ;**  
**double TotalCombinacoes ( int conjunto, int escolhas ) ;**

**Solução:**

**a) Arquivo "funcoes.cpp":**

```
#include <iostream>

// Calcula o fatorial de um número inteiro sem sinal.
// O resultado é unsigned long long, pois um fatorial pode não caber em 32 bits

unsigned long long Fatorial ( unsigned int num )
{
    unsigned long long result ;
    for ( result = 1 ; num > 1 ; --num )
        result *= num ; // instrução a executar se condição verdadeira

    // Laço para cálculo:
    // Inicia "result".
    // Avalia.
    // Executa [ result*=num ] (se e enquanto o número "num" for maior que 1).
    // Após a primeira execução (se ela chegou a ocorrer)
    // irá decrementar "num" antes de uma próxima avaliação.

    return result ;
}

// Calcula o resultado de "base" elevada a "exp", ambos inteiros sem sinal
// (para simplificar a função, pois um cálculo realista de Potencia deveria ser double)
unsigned long long Potencia ( unsigned int base, unsigned int exp )
{
    unsigned long long result ;
    for ( result = 1 ; exp > 0 ; --exp )
        result *= base ; // instrução a executar se condição verdadeira

    // Aplicam-se aqui os comentários sobre o laço da função acima (Fatorial)
    // trocando-se apenas as variáveis empregadas e a condição [ exp>0 ]

    return result ;
}

// Calcula o Total de Termos (e não a sua soma)
```

```
// de uma Progressão Aritmética:
int PA_TotalTermos ( int inicial, int final, int razao )
{
    // Inicialmente, testar se "razao" é diferente de zero
    // e também se é compatível com os valores de "inicial" e "final":

    /*
        // Solução com varios "ifs":

        if ( razao == 0 )
            return 0 ; // Progressão impossível.

        if ( inicial > final ) // inicial maior-que final
        {
            if ( razao > 0 ) // Razao deveria ser negativa
                return 0 ; // nada a fazer
        }
        else if ( inicial < final ) // inicial menor-que final
        {
            if ( razao < 0 ) // Razao deveria ser positiva
                return 0 ; // nada a fazer
        }
    */

    // Solução usando operadores lógicos:
    if ( razao == 0 || ( inicial > final && razao > 0 )
        || ( inicial < final && razao < 0 ) )
        return 0 ; // nada a fazer

    // Agora podemos efetuar o cálculo:
    // Eleva a precedência da subtração seguida de soma
    // e divide o resultado por "razao":
    return (final-inicial+razao) / razao ; // retorna o resultado da divisão
}


// Imprime lista de números pares:
void ImprimePares ( int inicial , int final )
{
    std::cout << "\nLista dos numeros pares entre "
        << inicial << " e " << final << '\n';

    // Descobrir se o inicial é ímpar.
    // Se for, não poderá ser impresso
    // Então deve ser incrementado, tornando-se par:

    // Primeira possibilidade de solução. Usa o operador de módulo:
    // se o resto da divisão por 2 for igual a 1, então o número é ímpar.
    // if ( inicial % 2 == 1 )
    //     ++inicial; // "inicial" torna-se par

    // Segunda possibilidade. Como queremos calcular o resto da divisão por 2, e
    // 2 é uma potência de 2, podemos usar o bitwise "and" entre "inicial" e 2-1 (1):
    // if ( ( inicial & 1 ) == 1 ) // se o resultado da divisão por 2 for 1
    //     ++inicial ; // "inicial" torna-se par

```

 Observar que no **"if"** acima, a **precedência** do *bitwise* **"and"** deve ser definida explicitamente: comparação por igualdade tem precedência mais alta.

```

// OK. Mas como o resultado só poderá ser ZERO ou UM,
// podemos fazer simplesmente:
if ( inicial & 1 ) // se o resultado for UM a expressão será verdadeira: é ímpar.

```

```

    ++inicial ; // "inicial" torna-se par
// Para saber quantos números serão impressos, chamar a função
// "PA_TotalTermos" com o valor 2 para "razao"
// já que essa é a distância entre números pares:
int total_pares = PA_TotalTermos( inicial, final, 2 ) ;
std::cout << "Total de numeros a imprimir: " << total_pares << '\n';

// Imprimir a lista de pares:
std::cout << "Pares no intervalo:\n";

if ( total_pares > 0 ) // se há o que imprimir:
{
    // Laço para imprimir os números pares no intervalo, separados por vírgulas:
    for ( int numero = inicial ; numero <= final ; numero += 2 )
        // "numero" só poderá ser usada neste laço...
    {
        if ( numero > inicial ) // se já é o segundo ou superior:
            std::cout << " , " ; // imprime branco-vírgula-branco antes

        // Imprime o número:
        std::cout << numero ;
    }

    std::cout << '\n'; // quebra de linha

    // Se tentássemos aqui:
    // std::cout << numero << '\n';
    // teríamos um erro de compilação pois "numero"
    // foi declarada no escopo do laço "for".
}
else
    std::cout << "Nenhum numero par nesse intervalo\n";

// E isso é tudo, pois:
// nenhum valor de retorno pode ser devolvido por esta função
// já que o valor de retorno foi especificado como "void".
}

// Ganhando 1 real no primeiro dia, dobrar o valor a cada dia
// até o dia "ultimo_dia":
double DobraValor (int ultimo_dia)
{
    // Fórmula de cálculo para o enésimo termo nesse tipo de Progressão Geométrica
    // - neste exemplo, o dia "ultimo_dia" (que pode ser 31 ou outro qualquer):
    /*
        [ an = a1 * q ^ (n-1) ] - (modelo genérico: o símbolo ^ simboliza
                                exponenciação - isso não é assim em C ou C++)

        Onde 'a1' é o primeiro termo, 'q' é a razão de multiplicação e 'n' é
        o total de termos.

        ou:
        an = a1 * Potencia( q , n-1 ) ;

        Neste caso (dobrar o valor a partir do dia 1), o primeiro termo é o dia 1 e a razão
        só pode ser 2 (a cada dia ganho o dobro do dia anterior);
        então podemos substituir:
        o 'a1' por 1;
        o 'q' por 2
        o e 'n' por 'ultimo_dia' (que é o parâmetro da função).
    */

    // Então, no último dia (o dia "ultimo_dia") ganharei:
    // 1 * Potencia( 2 , ultimo_dia - 1 );

```



```

// ou simplesmente:
return Potencia( 2 , ultimo_dia-1 ) ;
}

// O mesmo, usando um laço "for":
double DobraValor_for (int ultimo_dia )
{
    double result = 1;
    for ( int dia = 1; dia < ultimo_dia ; ++dia )
        result *= 2 ;

    // Multiplico o valor por 2, começando de 1, enquanto [ dia < ultimo_dia ]
    // Pois, se ultimo_dia == 1, ganho apenas 1
    return result ;
}

// Calcula a quantidade de combinações únicas, considerando uma quantidade de
// escolhas dentro de um conjunto de possibilidades:
unsigned long long TotalCombinacoes( unsigned int conjunto, unsigned int escolhas )
{
    return Fatorial ( conjunto ) /
           ( Fatorial(escolhas) * Fatorial(conjunto-escolhas) ) ;
}

```

#### b) Arquivo "main.cpp":

Neste arquivo implementaremos a função **main**. Ela deverá testar todas as funções do módulo anterior (**funcoes.cpp**). Para isso terá que chamá-las. Mas, como não estão visíveis **neste módulo**, o compilador não reconhecerá essas funções.

Precisamos portanto declarar os seus protótipos **antes** de efetuar as chamadas. Podemos fazer isso no topo do arquivo, antes da função **main**:

```

#include <iostream>

// Protótipos:
unsigned long long Fatorial ( unsigned int num ) ;
unsigned long long Potencia ( unsigned int base, unsigned int exp ) ;
int PA_TotalTermos ( int inicial, int final, int razao ) ;
void ImprimePares ( int inicial , int final ) ;
double DobraValor ( int ultimo_dia ) ;
double DobraValor_for ( int ultimo_dia ) ;
unsigned long long TotalCombinacoes ( unsigned int conjunto, unsigned int escolhas ) ;

```

Uma idéia **melhor** é colocar esses protótipos em um arquivo **header**. Pois, se quisermos usar essas funções em outros projetos, bastará **incluir** esse arquivo, ao invés de copiar e colar os protótipos.

Assim, podemos criar um **novo arquivo**, por exemplo **funcoes.h**, e inserir esses protótipos nesse arquivo:

**Arquivo "funcoes.h", contendo apenas os protótipos abaixo:**

```

unsigned long long Fatorial ( unsigned int num ) ;
unsigned long long Potencia ( unsigned int base, unsigned int exp ) ;
int PA_TotalTermos ( int inicial, int final, int razao ) ;
void ImprimePares ( int inicial , int final ) ;

```

```
double DobraValor ( int ultimo_dia ) ;
```

```
double DobraValor_for ( int ultimo_dia ) ;
```

```
unsigned long long TotalCombinacoes ( unsigned int conjunto, unsigned int escolhas ) ;
```

E agora fazer a inclusão no topo de **main.cpp**. Aliás a inclusão desse **header** seria útil no próprio arquivo **funcoes.cpp**. Pois, dentro desse módulo:

- A função **ImprimePares** chama a função **PA\_TotalTermos**.
- A função **DobraValor** chama a função **Potencia**.
- A função **TotalCombinacoes** chama a função **Fatorial**.

No momento não temos problemas pois as funções são chamadas **após sua implementação**. Mas isso significa que temos que nos preocupar com a ordem de escrita das funções. Poderíamos simplesmente esquecer esse assunto, fazendo uma inclusão no topo de **funcoes.cpp**.

O arquivo "**funcoes.cpp**", passa a ter estas duas linhas iniciais:

```
#include <iostream>
```

```
#include "funcoes.h" 
```

E o arquivo "**main.cpp**", ficaria assim:

```
#include <iostream>
```

```
#include <limits>
```

```
#include "funcoes.h" 
```

```
// A função abaixo é prototipada aqui, pois só sera usada neste módulo.
```

```
// O protótipo é necessario pois está implementada após sua chamada em "main"
```

```
// Analisa status de "cin". Em caso de erro,
```

```
// imprime mensagem e limpa flags de erro:
```

```
bool ValidarEntrada();
```

```
int main( )
```

```
{
```

```
    std::cout << "Testa Fatorial:\n";
```

```
    int num ;
```

```
    for ( num = 0 ; num < 5 ; ++num )
```

```
        std::cout << "Fatorial de " << num << " = "  

                                << Fatorial(num) << '\n';
```

```
    std::cout << "\nTesta Potencia:\n";
```

```
    for ( num = 0 ; num < 5 ; ++num )
```

```
        std::cout << "10 elevado a " << num << " = "  

                                << Potencia(10, num) << '\n';
```

```
    std::cout << "\nTesta PA_TotalTermos:\n";
```

```
    int razao;
```

```
    // Testa com inicial menor que final, razao positiva:
```

```
    for ( razao = 0 ; razao < 4 ; ++razao )
```

```
        std::cout << "Total termos entre 1 e 10, razao " << razao << " = "  

                                << PA_TotalTermos(1, 10, razao) << '\n';
```

```
    // Testa com inicial maior que final, razao negativa:
```

```
    for ( razao = -1 ; razao >= -4 ; --razao )
```

```
        std::cout << "Total termos entre 10 e 1, razao " << razao << " = "  

                                << PA_TotalTermos(10, 1, razao) << '\n';
```

```
    // Testa com inicial menor que final, razao negativa:
```

```
    std::cout << "Total termos entre 1 e 10, razao " << -1 << " = "
```

```

    << PA_TotalTermos(1, 10, -1) << '\n';
// Testa com inicial maior que final, razao positiva:
std::cout << "Total termos entre 10 e 1, razao " << 1 << " = "
    << PA_TotalTermos(10, 1, 1) << '\n';
// Testa com inicial igual a final:
std::cout << "Total termos entre 10 e 10, razao " << 1 << " = "
    << PA_TotalTermos(10, 10, 1) << '\n';
std::cout << "Total termos entre 10 e 10, razao " << -1 << " = "
    << PA_TotalTermos(10, 10, -1) << '\n';

std::cout << "\nTesta ImprimePares:\n";

ImprimePares(1,11); // inicial ímpar, final ímpar
ImprimePares(1,10); // inicial ímpar, final par
ImprimePares(2,11); // inicial par, final ímpar
ImprimePares(2,10); // inicial par, final par
ImprimePares(12,12); // iguais e pares
ImprimePares(11,11); // iguais e ímpares
ImprimePares(12,11); // inicial maior-que final

std::cout << "\nTesta DobraValor:\n";

std::cout << "Ganhando 1 real no primeiro dia de um mes,\n"
    << "e dobrando o valor todos os dias,\n"
    << "no dia 31 ganharei: \n";
std::cout << int(DobraValor(31)) << "\n";

std::cout << "O mesmo calculo usando o 'for': \n";
std::cout << int(DobraValor_for(31)) << "\n";

std::cout << "\nTesta TotalCombinacoes:\n";
std::cout << "Usando Fatorial, sei as chances de ganhar:";
std::cout << "\nNa MegaSena : 1 em "
    << int( TotalCombinacoes(60, 6) ) ;
std::cout << "\nNa LotoFacil: 1 em "
    << int ( TotalCombinacoes(25, 15) ) ;

// Testando com valores informados pelo usuário.
std::cout << "\n\nSolicitando valores ao usuario\n";
std::cout << "\nInforme numero para calculo de fatorial: ";
std::cin >> num;
if ( ValidarEntrada() )
    std::cout << "Fatorial de " << num << " = "
        << Fatorial(num) << '\n';

std::cout << "\nInforme base e expoente para calculo de Potencia: ";
int base, exp;
std::cin >> base >> exp ;
if ( ValidarEntrada() )
    std::cout << base << " elevado a " << exp << " = "
        << Potencia(base, exp) << '\n';

std::cout << "\nInforme inicial, final e razao p/calculo"
    << " do total de termos da PA: " ;
int inicial, final ;
std::cin >> inicial >> final >> razao ;
if ( ValidarEntrada() )
    std::cout << "\nTotal de termos nessa PA: " << " = "
        << PA_TotalTermos(inicial, final, razao) << '\n';

std::cout << "\nInforme inicial e final p/lista de pares: ";

```

```

std::cin >> inicial >> final ;
if ( ValidarEntrada() )
    ImprimePares( inicial, final );
return 0 ;
}

// Analisa status de "cin". Em caso de erro,
// imprime mensagem e limpa flags de erro:
bool ValidarEntrada()
{
    if ( std::cin.fail() )
    {
        std::cout << "valores incorretos\n";
        // Limpa erros:
        std::cin.clear();
        // Ignora "new-lines" pendentes no buffer:
        std::cin.ignore( std::numeric_limits<int>::max() , '\n' );
        return false; // entrada inválida
    }
    return true; // entrada válida
}

```

## 2. Exercício 2

Este exercício foi proposto na seção 5.11.3, página 179.

---

**Enunciado:** Implemente uma **calculadora** que realize as 4 operações básicas (soma, subtração, divisão e multiplicação).

---

- Use dois módulos. O primeiro deles (**main.cpp**) deverá conter a função **main**, a qual irá testar as funções do segundo módulo. No segundo módulo (**calculadora.cpp**), escreva duas versões da função **Calculadora**. As duas irão pedir ao usuário, **dentro de um laço**, que informe: o **primeiro operando**, o **operador** e o **segundo operando**.
    - Após cada operação, perguntar se deve ser feita nova operação.
    - A única diferença entre essas duas funções é que uma usará [ **if else** ] para analisar o **operador** e a segunda usará [ **switch ( operador )** ].
- ```

void Calculadora_if_else ( ) ;
void Calculadora_switch ( ) ;

```

**Solução:**

**a) Arquivo "calculadora.h", contendo estes protótipos:**

```

void Calculadora_if_else( ) ;
void Calculadora_switch( ) ;

```

**b) Arquivo "calculadora.cpp":**

```

#include <iostream>
#include <limits>

```

```

// A função abaixo é prototipada aqui, pois só sera usada neste módulo.
// O protótipo é necessario pois está implementada após sua chamada em "main"
// Analisa status de "cin". Em caso de erro,
// imprime mensagem e limpa flags de erro:

```

**bool ValidarEntrada();**

```
// ==== Calculadora que usa [ if else ]
void Calculadora_if_else( )
{
    bool continuar = true;
    while ( continuar )
    {
        int operando_1 , operando_2 ;
        char operador ;
        std::cout << "\nInforme operando_1, operador e operando_2: " ;

        // solicita dados:
        std::cin >> operando_1 >> operador >> operando_2 ;

        // Checa se dados estão corretos:
        if ( ValidarEntrada() )
        {
            if ( operador == '+' )
                std::cout << "Somar: " << operando_1 + operando_2 << '\n' ;

            else if ( operador == '-' )
                std::cout << "Subtrair: " << operando_1 - operando_2 << '\n' ;

            else if ( operador == '*' )
                std::cout << "Multiplicar: " << operando_1 * operando_2
                                                                << '\n' ;

            else if ( operador == '/' )
                std::cout << "Dividir: " << operando_1 / operando_2 << '\n' ;

            else
                std::cout << "operador invalido\n";

            std::cout << "Nova operacao? (0)encerrar, (1)continuar: ";
            std::cin >> continuar ; // isto poderá levar à interrupção do laço

            if ( !ValidarEntrada() ) // entrada incorreta aqui:
                continuar = false; // fim: interrompe laço

        } // fim do primeiro "if ( ValidarEntrada ( ) )"
    } // fim do "while"
} // fim da função "Calculadora_if_else( )"

// ==== Calculadora que usa [ switch ]
void Calculadora_switch( )
{
    bool continuar = true;
    while ( continuar )
    {
        int operando_1 , operando_2 ;
        char operador ;
        std::cout << "\nInforme operando_1, operador e operando_2: " ;

        // solicita dados:
        std::cin >> operando_1 >> operador >> operando_2 ;

        // Checa se dados estão corretos:
        if ( ValidarEntrada() )
        {
```

```

switch ( operador )
{
    case '+' :
        std::cout << "Somar: " << operando_1 + operando_2
        << '\n' ;
        break;
    case '-' :
        std::cout << "Subtrair: " << operando_1 - operando_2
        << '\n' ;
        break;
    case '*' :
        std::cout << "Multiplicar: " << operando_1 * operando_2
        << '\n' ;
        break;
    case '/' :
        std::cout << "Dividir: " << operando_1 / operando_2
        << '\n' ;
        break;
    default:
        std::cout << "operador invalido\n";
} // fim do "switch"

std::cout << "Nova operacao? (0)encerrar, (1)continuar: ";
std::cin >> continuar ; // isto podera levar à interrupção do laço
if ( !ValidarEntrada() ) // entrada incorreta aqui:
    continuar = false; // fim: interrompe laço
} // fim do primeiro "if ( ValidarEntrada ( ) )"
} // fim do "while"
} // fim da função "Calculadora_switch( )"

// Analisa status de "cin". Em caso de erro,
// imprime mensagem e limpa flags de erro:
bool ValidarEntrada()
{
    if ( std::cin.fail() )
    {
        std::cout << "valores incorretos\n";
        // Limpa erros:
        std::cin.clear();
        // Ignora "new-lines" pendentes no buffer:
        std::cin.ignore( std::numeric_limits<int>::max() , '\n' );
        return false; // entrada inválida
    }
    return true; // entrada válida
}

```

### c) Arquivo "main.cpp":

```

#include <iostream>
#include "calculadora.h"

int main( )
{
    // 1) Testa "Calculadora_if_else"
    std::cout << "Testa 'Calculadora_if_else'\n";
}

```

```
Calculadora_if_else( );  
// 2) Testa "Calculadora_switch"  
std::cout << "\nTesta 'Calculadora_switch'\n";  
Calculadora_switch( );  
return 0 ;  
}
```

### 3. Questões

Estas questões foram propostas na seção **5.11.4**, página **180**.

**Cap.5 - 1.** Assinale as afirmações verdadeiras:

- a. ☐ O comentário iniciado por */\** e finalizado por *\*/* só é válido em **C**.  
{  
    **Errado.** É válido também em **C++**.  
}
- b. ☐ O comentário iniciado por */\** e finalizado por *\*/* só é válido em **C++**.  
{  
    **Errado.** É válido também em **C**.  
}
- c. ☒ O comentário iniciado por */\** e finalizado por *\*/* é válido em **C** e em **C++**.
- d. ☒ O comentário iniciado por *//* e finalizado pela quebra de linha do texto (*new-line*) só é válido em **C++** ou em **C99**, embora muitos compiladores, dependendo das opções de compilação, o aceitem para **C89**.

**Cap.5 - 2.** Em uma linha de instrução podemos:

- a. ☐ Encadear diversas operações, inclusive chamadas de função, independentemente dos retornos dessas funções.  
{  
    **Errado.** O **tipo do retorno das funções** deve ser compatível com as operações envolvidas. Por exemplo, atribuir o retorno de uma função cujo valor de retorno é **double** a uma variável do tipo **int**, pode levar à truncagem do valor retornado. Outro exemplo de incompatibilidade está na resposta 'c', abaixo.  
}
- b. ☒ Combinar operações, inclusive chamadas de funções, desde que os retornos das funções, usadas como operandos, sejam compatíveis com os operadores associados.
- c. ☐ Atribuir a uma variável o retorno de uma função com valor de retorno **void**.  
{  
    **Errado.** O **valor de retorno void** indica retorno "vazio" ou inexistência de um valor de retorno. Logo, não pode haver atribuição do retorno de funções assim a nenhuma variável, já que **não existem** variáveis do tipo **void** (vazias).  
}

**Cap.5 - 3.** Assinale as afirmações corretas, considerando o código abaixo:

```
char c ; for ( c = 2 ; c >= 0 ; ++c ) { std::cout << c ; }
```

- a. ☐ O código acima leva a um erro de compilação.  
{  
  **Errado.** Nenhum erro de sintaxe foi cometido nesse código.  
}
- b. ☐ O bloco de instruções associado ao laço será executado **2 vezes**.  
{  
  **Errado.** No laço, a variável 'c' é iniciada com **2**. Em seguida, ocorre o teste de condição [ **c >= 0** ]. O valor **2** é maior-que zero, e a avaliação resulta **verdadeira**. Então, em seguida o valor de 'c' será impresso. Agora, 'c' será incrementada (passando para **3**). Continuará assim maior que zero, o mesmo ocorrendo no próximo incremento (**4**). Será executada mais do que duas vezes.  
}
- c. ☐ O bloco de instruções associado ao laço será executado **uma vez**.  
{  
  **Errado.** Ver a resposta 'b', acima.  
}
- d. ☐ Será executado infinitamente (**laço infinito**).  
{  
  **Errado.** Esta afirmação **estaria correta** se o teste de condição fosse [ **c >= 0 || c < 0** ]. Mas o que temos é [ **c >= 0** ]. Como o tipo de 'c' é **char**, o seu valor máximo é **127**. Ao atingir esse valor, ocorrerá um estouro e o valor de 'c' será **-128** (valor **negativo**). Logo, estará **menor-que zero** e com isso a avaliação da condição terá resultado **falso**, interrompendo o laço.  
}
- e. ☐ **Nunca** será executado.  
{  
  **Errado.** No laço, a variável 'c' é iniciada com **2**. Em seguida, ocorre o teste de condição [ **c >= 0** ]. O valor **2** é maior-que zero. Logo, será executada no mínimo uma vez. Ver também a resposta 'b', acima.  
}
- f. ☒ **Nenhuma** das respostas acima está correta.  
{ Pois o laço será executado bem mais do que duas vezes, mas será interrompido quando, após atingir 127, receber novo incremento, tornando-se negativo: **-128**. Ver também a resposta 'd', acima. }

**Cap.5 - 4.** Assinale as afirmações corretas, considerando o código abaixo:

```
unsigned char uc ; for ( uc = 2 ; uc >= 0 ; ++uc )  
{ std::cout << uc ; }
```

- a. ☐ O código acima leva a um erro de compilação.  
{  
  **Errado.** Nenhum erro de sintaxe foi cometido nesse código.  
}
- b. ☐ O bloco de instruções associado ao laço será executado **2 vezes**.  
{



- 
- Errado.** A resposta '3.b', acima, também é válida aqui.
- 
- c. ☐ *O bloco de instruções associado ao laço será executado **uma vez**.*
- {
- Errado.** A resposta '3.b', acima, também é válida aqui.
- 
- d. ☒ **Será executado infinitamente (laço infinito).**
- { Pois, como o tipo de 'uc' é **unsigned char**, seu valor máximo é **255**. Após atingir esse valor, receberá novo incremento e ocorrerá um estouro, passando para **zero**. Como a condição é [ **uc >= 0** ], o resultado da avaliação será **sempre verdadeiro**. }
- 
- e. ☐ **Nunca será executado.**
- {
- Errado.** No mínimo uma vez, já que inicia 'uc' com **2** e o teste de condição é [ **uc >= 0** ]. Ver a resposta '3.b', acima: também é válida aqui.
- 
- f. ☐ **Nenhuma das respostas acima está correta.**
- {
- Errado.** A resposta 'd' está **correta**.
- 

**Cap.5 - 5.** Assinale as afirmações corretas, considerando o código abaixo:

```
char c; for ( c = 2; c >= 0 || c < 0; ++c ) { std::cout << c; }
```

- 
- a. ☐ *O código acima leva a um erro de compilação.*
- {
- Errado.** Nenhum erro de sintaxe foi cometido nesse código.
- 
- b. ☐ *O bloco de instruções associado ao laço será executado **2 vezes**.*
- {
- Errado.** Observar que a condição [ **c >= 0 || c < 0** ] só pode levar a **um laço infinito**. Qualquer que seja o valor de 'c' ele será **sempre [ maior-ou igual-a zero ] ou [ menor-que zero ]**. Logo será executado infinitamente.
- 
- c. ☐ *O bloco de instruções associado ao laço será executado **uma vez**.*
- {
- Errado.** Ver a resposta 'b', acima.
- 
- d. ☒ **Será executado infinitamente (laço infinito).**
- 
- e. ☐ **Nunca será executado.**
- {
- Errado.** Ver a resposta 'b', acima.
- 
- f. ☐ **Nenhuma das respostas acima está correta.**
- {
- Errado.** A resposta 'd' está **correta**.
-

---

}

---

**Cap.5 - 6.** Assinale as afirmações corretas, considerando o código abaixo:

```
int x ; for ( x=10 ; x > 1 ; --x ) { std::cout << x ; }
```

O bloco de instruções associadas ao laço for será executado:

- a. ☐ **11 vezes.**  
 {  
   **Errado.** Imprimirá de **10 a 2: 10, 9, 8, 7, 6, 5, 4, 3, 2.**  
   Pois 'x' é iniciado com **10** e a condição é [ **x > 1** ], sendo que após a execução da impressão, ocorre um **decremento**.  
   Logo, o bloco de instruções será executado **9 vezes**, o que poderia ser comprovado por uma chamada à função **PA\_TotalTermos** (que vimos no exercício **06\_dois modulos**): **PA\_TotalTermos( 10, 2, -1 )**.  
 }
- b. ☐ **10 vezes.**  
 {  
   **Errado.** Ver a resposta 'a', acima.  
 }
- c. ☒ **9 vezes**, de acordo com o resultado de uma chamada à função "PA\_TotalTermos", implementada no exercício "06\_dois\_modulos", acima: [ **PA\_TotalTermos ( 10 , 2 , -1 ) ;** ].
- d. ☐ *Infinitamente (laço infinito).*  
 {  
   **Errado.** Ver a resposta 'a', acima.  
 }
- e. ☐ *Nunca será executado.*  
 {  
   **Errado.** Ver a resposta 'a', acima.  
 }
- f. ☐ *Após o encerramento do for o valor de "x" será zero.*  
 {  
   **Errado.** Se a condição é [ **x > 1** ] o laço será interrompido quando 'x' atingir **1**, e com isso a avaliação da condição terá resultado **falso**. Assim, dentro do laço o último valor a ser impresso será **2 (x>1)**, e **após o laço**, o valor de 'x' será **1**.  
 }
- g. ☒ Após o encerramento do **for** o valor de "x" será **um**.
- h. ☐ *Após o encerramento do for o valor de "x" será dois.*  
 {  
   **Errado.** Ver a resposta 'f', acima.  
 }

**Cap.5 - 7.** Assinale as afirmações corretas, considerando o código abaixo:

```
int x ; std::cin >> x ; if ( x == 10 ) { ++x ; }
```

// atenção para o símbolo em ( x == 10 ): dois sinais de igual

Quando o 'if' acima **for executado**, a variável 'x' será **incrementada**:

- a. ☐ **10 vezes, se 'x' for igual a 10..**

- 
- {**  
**Errado.** Não se aplica. Um teste de condição **if não é um laço**. Como há **uma única instrução** de incremento associada ao **if** (e não dez delas), no máximo haverá **um único incremento** se a avaliação da condição **[ x == 10 ]** tiver resultado **verdadeiro** (se o valor de 'x' for **10**). Do contrário, não haverá nenhum incremento.  
**}**
- 
- b. ☐ **Sempre será incrementada uma vez.**  
**{**  
**Errado.** A variável 'x' será incrementada uma vez **sempre que** a avaliação da condição **[ x == 10 ]** tiver resultado **verdadeiro** (se o valor de 'x' for **10**). **Do contrário**, não haverá **nenhum incremento**.  
**}**
- 
- c. ☒ **Uma vez, somente se 'x' for igual a 10.**
- 
- d. ☐ **Nunca será incrementada.**  
**{**  
**Errado.** Depende da avaliação da condição. Ver resposta 'b', acima.  
**}**
- 
- e. ☐ **A operação é incorreta e ocorrerá um erro de compilação.**  
**{**  
**Errado.** Nenhum erro de sintaxe foi cometido nesse código.  
**}**
- 
- f. ☐ **Nenhuma das respostas acima está correta.**  
**{**  
**Errado.** A resposta 'c' está **correta**.  
**}**
- 

**Cap.5 - 8.** Assinale as afirmações corretas, considerando o código abaixo:

```
int x; std::cin >> x; if ( x = 10 ) { ++x; }
```

// atenção para o símbolo em ( x = 10 ): **um** sinal de igual

Quando o 'if' acima **for executado**, a variável 'x' será **incrementada**:

- 
- a. ☐ **10 vezes.**  
**{**  
**Errado.** Não se aplica. **Não é um laço** e, no bloco associado ao **if**, há um único incremento (e não dez deles).  
**}**
- 
- b. ☐ **Uma vez, somente se 'x' for igual a 10.**  
**{**  
**Errado.** Observar que a condição é **[ x = 10 ]** (um sinal de igual) e **não [ x == 10 ]** (dois sinais de igual). Isto significa que **antes da avaliação ocorre uma atribuição**. Isto é, independentemente do valor anterior de 'x', nessa expressão **10 é copiado para 'x'**. Logo 'x' passa a conter **10**, e **em seguida é avaliado**: como **10 é diferente de zero** (diferente de falso) a avaliação **sempre** terá resultado **verdadeiro**. Em consequência, 'x' **sempre será incrementada**.  
**}**
- 
- c. ☐ **Nunca será incrementada.**
-

---

```
{
    Errado. Ver resposta 'b', acima.
}
```

---

d. ☒ **Sempre** será incrementada.

---

e. ☐ A operação é **incorreta** e ocorrerá um erro de compilação.

```
{
    Errado. Nenhum erro de sintaxe foi cometido nesse código.
}
```

---

f. ☐ Nenhuma das respostas acima está correta.

```
{
    Errado. A resposta 'd' está correta.
}
```

---

**Cap.5 - 9.** Assinale as afirmações corretas, considerando o código abaixo:

```
int x ; std::cin >> x ; if ( x > 20 && x < 21 ) { ++x; }
```

Quando o 'if' acima **for executado**, a variável 'x' será **incrementada**:

---

a. ☒ **Nunca** será incrementada.

---

b. ☐ **Sempre** será incrementada.

```
{
    Errado. Não existem números inteiros que sejam [ maiores que 20 ] e [ menores que 21 ].
    Se maior que 20 será no mínimo 21, logo não será menor que 21.
    E se menor que 21 será no máximo 20, logo não será maior que 20).
    Seria diferente se 'x' fosse double ou float: neste caso poderíamos ter 20.1, 20.2, etc.
    Então, do modo como está formulada esta questão, a variável 'x' nunca será incrementada.
}
```

---

c. ☐ Será incrementada sempre que 'x' for **igual a 20**.

```
{
    Errado. Ver a resposta 'b', acima.
}
```

---

d. ☐ Será incrementada sempre que 'x' for **igual a 21**.

```
{
    Errado. Ver a resposta 'b', acima.
}
```

---

e. ☐ Será incrementada se 'x' for [ **maior que 20** ] OU [ **menor que 21** ].

```
{
    Errado. A condição é [ x > 20 ] e [ x < 21 ]. Isto é temos um operador lógico and, e, assim, as duas operações relacionais precisam retornar um resultado verdadeiro.
    Para que esta resposta fosse correta, a condição teria que ser:
    [ x > 20 || x < 21 ], ou seja: teríamos um operador lógico or conectando as duas operações relacionais. Nesse caso, bastaria que uma delas fosse verdadeira.
}
```

---

f. ☐ A operação é **incorreta** e ocorrerá um erro de compilação.

---

---

```
{
  Errado. Nenhum erro de sintaxe foi cometido nesse código.
}
```

---

g. ☐ *Nenhuma das respostas acima está correta.*

```
{
  Errado. A resposta 'a' está correta.
}
```

---

**Cap.5 - 10.** Assinale as afirmações corretas, considerando o código abaixo:

```
int x ; std::cin >> x ; if ( x > 20 || x < 21 ) { ++x; }
```

Quando o 'if' acima **for executado**, a variável 'x' será **incrementada**:

a. ☐ *Nunca será incrementada.*

```
{
  Errado. Qualquer número inteiro pode ser [ maior que 20 ] ou [ menor que 21 ]. No limite, se 'x' for 20 será menor que 21; e se for 21, será maior que 20.
  Observar que a condição é [ x > 20 ] ou [ x < 21 ]. Logo, as duas operações relacionais estão conectadas pelo operador lógico or, e assim basta que uma delas seja verdadeira para que a avaliação retorne resultado verdadeiro.
  Então, a variável 'x' sempre será incrementada.
}
```

---

b. ☒ **Sempre** será incrementada.

c. ☐ *Será incrementada **somente se 'x' for igual a 20**.*

```
{
  Errado. Ver a resposta 'a' acima.
}
```

---

d. ☐ *Será incrementada **somente se 'x' for igual a 21**.*

```
{
  Errado. Ver a resposta 'a' acima.
}
```

---

e. ☐ *A operação é **incorreta** e ocorrerá um erro de compilação.*

```
{
  Errado. Nenhum erro de sintaxe foi cometido nesse código.
}
```

---

f. ☐ *Nenhuma das respostas acima está correta.*

```
{
  Errado. A resposta 'b' está correta.
}
```

---

**Cap.5 - 11.** Assinale as afirmações corretas, considerando o código abaixo:

```
int x ; std::cin >> x ;
if ( ( x > 20 || x < 21 ) && ( x > 20 && x < 21 ) ) { ++x; }
```

Quando o 'if' acima **for executado**, a variável 'x' será **incrementada**:

a. ☒ **Nunca** será incrementada.

b. ☐ **Sempre** será incrementada.

---

---

```
{
Errado. Temos uma operação lógica and conectando:
- uma operação lógica or [ x > 20 || x < 21 ]
- uma operação lógica and [ x > 20 && x < 21 ]
A primeira operação (or) terá sempre resultado verdadeiro (ver a resposta '10.a', acima).
Já a segunda operação (and) terá sempre resultado falso (ver a resposta '9.b', acima).
Em consequência a primeira operação terá sempre resultado falso e a segunda sempre verdadeiro. Como a expressão completa é:
{ ( [ x > 20 ] ou [ x < 21 ] ) e ( [ x > 20 ] e [ x < 21 ] ) }, as duas deveriam ser verdadeiras para que o resultado final da avaliação fosse verdadeiro. Como uma delas será sempre falsa, o resultado final também será sempre falso. Então, a variável 'x' nunca será incrementada.
}
```

---

c. ☐ *Será incrementada sempre que '**x**' for **igual a 20**.*

```
{
Errado. Ver resposta 'b', acima.
}
```

---

d. ☐ *Será incrementada sempre que '**x**' for **igual a 21**.*

```
{
Errado. Ver resposta 'b', acima.
}
```

---

e. ☐ *A operação é **incorreta** e ocorrerá um erro de compilação.*

```
{
Errado. Nenhum erro de sintaxe foi cometido nesse código.
}
```

---

f. ☐ *Nenhuma das respostas acima está correta.*

```
{
Errado. A resposta 'a' está correta.
}
```

---

**Cap.5 - 12.** Assinale as afirmações corretas, considerando o código abaixo:

```
int x, y; std::cin >> x >> y;
if ( ( x > 20 && x < 22 ) && ( y > 20 && y < 22 ) ) { ++x; }
```

Quando o 'if' acima **for executado**, a variável '**x**' será **incrementada**:

a. ☐ ***Nunca** será incrementada.*

```
{
Errado. Temos uma operação lógica and conectando duas operações lógicas and. Ambas poderão ter resultados verdadeiros dependendo dos valores de 'x' e 'y'.
Pois, tanto para 'x' como para 'y', é exigido que o valor seja [ maior que 20 ] e [ menor que 22 ].
E existe um número inteiro entre 20 e 22: o número 21.
Logo, se [ 'x' for 21 ] e [ 'y' for 21 ] as duas operações terão resultado verdadeiro e o resultado final também será verdadeiro.
}
```

---

- b. ☐ **Sempre** será incrementada.  
 {  
     **Errado.** Depende dos valores de 'x' e de 'y'. Ver resposta 'a', acima.  
 }
- c. ☐ Sempre que 'x' for **igual a 21**; 'y' poderá conter **20, 21 e 22**.  
 {  
     **Errado.** A variável 'y' deve ser [ **maior que 20** ] e [ **menor que 22** ].  
     Ver também a resposta 'a', acima.  
 }
- d. ☐ Sempre que 'y' for **igual a 21**; 'x' poderá conter **20, 21 e 22**.  
 {  
     **Errado.** A variável 'x' deve ser [ **maior que 20** ] e [ **menor que 22** ].  
     Ver também a resposta 'a', acima.  
 }
- e. ☒ Sempre que [ 'x' for **igual a 21** ] **E** [ 'y' for **igual a 21** ].
- f. ☐ A operação é **incorreta** e ocorrerá um erro de compilação.  
 {  
     **Errado.** Nenhum erro de sintaxe foi cometido nesse código.  
 }
- g. ☐ Nenhuma das respostas acima está correta.  
 {  
     **Errado.** A resposta 'e' está **correta**.  
 }

**Cap.5 - 13.** Assinale as afirmações corretas, considerando o código abaixo:

```
int x ; cin >> x ; while ( true ) { ++x ; }
```

A cada vez que o '**while**' acima for executado, a variável '**x**' será **incrementada**:

- a. ☐ **Uma vez.**  
 {  
     **Errado.** A condição estabelecida para avaliação no laço **while** é, simplesmente: [ **true** ].  
     Isto é, o que será avaliado é uma constante (inalterável). E o resultado da avaliação será **sempre verdadeiro**, pois **true** é verdadeiro.  
     Logo, a variável '**x**' será **incrementada infinitamente** (laço infinito).  
 }
- b. ☐ **Nunca** será incrementada.  
 {  
     **Errado.** Ocorrerá exatamente o oposto. Ver resposta 'a', acima.  
 }
- c. ☒ Será incrementada **infinitamente** (laço infinito).
- d. ☐ A operação é **incorreta** e ocorrerá um erro de compilação.  
 {  
     **Errado.** Nenhum erro de sintaxe foi cometido nesse código.  
 }

- 
- e. ☐ *Nenhuma das respostas acima está correta.*
- ```
{
    Errado. A resposta 'c' está correta.
}
```
- 

**Cap.5 - 14.** Assinale as afirmações corretas, considerando o código abaixo:

```
int x ; cin >> x ; while ( false ) { ++x ; }
```

A cada vez que o **'while'** acima for executado, a variável **'x'** será **incrementada**:

- 
- a. ☐ *Uma vez.*
- ```
{
    Errado. A condição estabelecida para avaliação no laço while é,
    simplesmente: [ false ].
    Isto é, o que será avaliado é uma constante (inalterável). E o resulta-
    do da avaliação será sempre falso, pois false é falso.
    Logo, a variável 'x' nunca será incrementada.
}
```
- 
- b. ☒ **Nunca** será incrementada.
- 
- c. ☐ *Será incrementada **infinitamente** (laço infinito).*
- ```
{
    Errado. Ocorrerá exatamente o oposto. Ver resposta 'a', acima.
}
```
- 
- d. ☐ *A operação é **incorreta** e ocorrerá um erro de compilação.*
- ```
{
    Errado. Nenhum erro de sintaxe foi cometido nesse código.
}
```
- 
- e. ☐ *Nenhuma das respostas acima está correta.*
- ```
{
    Errado. A resposta 'b' está correta.
}
```
- 

**Cap.5 - 15.** Assinale as afirmações corretas, considerando o código abaixo:

```
int y ; cin >> y ; int x = y > 5 && y < 7 ;
```

- 
- a. ☐ *'x' poderá armazenar o número inteiro 5 ou o número inteiro 7.*
- ```
{
    Errado. O resultado de operações relacionais e lógicas é sempre lógico (ou booleano). Assim, [ y > 5 ] retornará true ou false, do mesmo modo que [ y < 7 ].
    E a avaliação final da expressão completa, baseada no operador lógico and { [ y > 5 ] e [ y < 7 ] }, também terá resultado lógico: true se as duas operações relacionais forem verdadeiras e false se uma delas for falsa.
    O resultado da avaliação(true ou false) será finalmente convertido para int [ int x = ... ], assumindo os valores 1 (verdadeiro) ou zero (falso).
    Neste caso, 'x' receberá o valor 1 se 'y' for igual a 6, pois esse é o único número inteiro que é [ maior que 5 ] e [ menor que 7 ]. Do contrário, receberá o valor 0(zero).
}
```
-



- 
- }
- 
- b. ☐ 'x' sempre armazenará o valor inteiro 1.  
 {  
   **Errado.** Poderá armazenar **1 ou 0**, dependendo do valor de 'y'.  
   Ver resposta 'a', acima.  
 }
- 
- c. ☐ 'x' sempre armazenará o valor inteiro 0.  
 {  
   **Errado.** Poderá armazenar **0 ou 1**, dependendo do valor de 'y'.  
   Ver resposta 'a', acima.  
 }
- 
- d. ☒ 'x' armazenará o valor inteiro 1 se 'y' for igual a 6.
- 
- e. ☒ 'x' armazenará o valor inteiro 0 se 'y' for diferente de 6.
- 
- f. ☐ A operação é **incorreta** e ocorrerá um erro de compilação.  
 {  
   **Errado.** Nenhum erro de sintaxe foi cometido nesse código.  
 }
- 
- g. ☐ Nenhuma das respostas acima está correta.  
 {  
   **Errado.** As respostas 'd' e 'e' estão **corretas**.  
 }
- 

**Cap.5 - 16.** Assinale as afirmações corretas, considerando o código abaixo:

```
int y; cin >> y; bool x = y > 5 && y < 7;
```

- 
- a. ☐ 'x' poderá armazenar o número inteiro 5 ou o número inteiro 7.  
 {  
   **Errado.** Em primeiro lugar, a variável 'x' é do **tipo bool** (e não int). Logo, os valores que pode armazenar são **true** ou **false**. Além disso, como vimos na questão acima, o resultado de operações relacionais e lógicas é sempre **lógico** (ou *booleano*). Assim, [ **y > 5** ] retornará **true** ou **false**, **do mesmo modo que** [ **y < 7** ]. E a avaliação final da expressão completa, baseada no operador lógico **and** { [ **y > 5** ] **e** [ **y < 7** ] }, também terá resultado lógico: **true** se as duas operações relacionais forem verdadeiras e **false** se uma delas for falsa. Neste caso, 'x' receberá o valor **true** se 'y' for **igual a 6**, pois esse é o único número **inteiro** que é [ **maior que 5** ] **e** [ **menor que 7** ]. Do contrário, receberá o valor **false**.  
 }
- 
- b. ☐ 'x' sempre armazenará o valor booleano **true**.  
 {  
   **Errado.** Poderá armazenar **true ou false**, dependendo do valor de 'y'. Ver resposta 'a', acima.  
 }
- 
- c. ☐ 'x' sempre armazenará o valor booleano **false**.  
 {  
   **Errado.** Poderá armazenar **false ou true**, dependendo do valor de
-

- 
- 'y'. Ver resposta 'a', acima.  
}
- 
- d. ☒ 'x' armazenará o valor *booleano true* se 'y' for igual a 6.
- 
- e. ☒ 'x' armazenará o valor *booleano false*, se 'y' for diferente de 6.
- 
- f. ☐ A operação é *incorreta* e ocorrerá um erro de compilação.  
{  
    **Errado.** Nenhum erro de sintaxe foi cometido nesse código.  
}
- 
- g. ☐ Nenhuma das respostas acima está correta.  
{  
    **Errado.** As respostas 'd' e 'e' estão **corretas**.  
}
- 

**Cap.5 - 17.** Assinale as afirmações corretas, considerando o código abaixo:

```
int y ; cin >> y ; bool x = y <= 5 || y >= 7 ;
```

- 
- a. ☐ 'x' sempre armazenará o valor *booleano true*.  
{  
    **Errado.** Dependendo do valor de 'y', a variável 'x' poderá armazenar **true** ou **false**. Neste caso, 'y' deverá ser:  
    [ **menor-ou-igual-a 5** ] ou [ **maior-ou-igual-a 7** ].  
    O único número inteiro que **não atende** a essa condição é o número **6**: pois nem é menor-ou-igual-a 5, nem é maior-ou-igual-a 7, logo, nesse caso, as duas operações relacionais têm resultado falso. O resultado da avaliação final é também **falso**, já que **pelo menos uma** das duas deveria ser verdadeira (**or**).  
    Assim, se 'y' contiver **6**, o resultado da avaliação será **false**, do **contrário**, **true**. E esse resultado será atribuído a 'x'.  
}
- 
- b. ☐ 'x' sempre armazenará o valor *booleano false*.  
{  
    **Errado.** Ver a resposta 'a', acima.  
}
- 
- c. ☒ 'x' armazenará o valor *booleano true* se 'y' for diferente de 6.
- 
- d. ☒ 'x' armazenará o valor *booleano false*, se 'y' for igual a 6.
- 
- e. ☐ A operação é *incorreta* e ocorrerá um erro de compilação.  
{  
    **Errado.** Nenhum erro de sintaxe foi cometido nesse código.  
}
- 
- f. ☐ Nenhuma das respostas acima está correta.  
{  
    **Errado.** As respostas 'c' e 'd' estão **corretas**.  
}
- 

**Cap.5 - 18.** Assinale as afirmações corretas, considerando o código abaixo:

```
// ...  
int x ;
```

```
std::cin >> x ;
switch ( x )
{
    case 1 :
        funcao1 ( );
        break;

    case 2 :
        funcao2 ( );

    case 3 :
        funcao3 ( );
        return;

    case 4 :
        funcao4 ( );

    default :
        funcao5 ( );
}
++x ;
```

- a. ☒ se 'x' for **igual a 1**, a **funcao1** será chamada e em seguida 'x' será incrementada.
- b. ☐ se 'x' for **igual a 2**, a **funcao2** será chamada e em seguida 'x' será incrementada.
- {  
    **Errado.** Como, após a chamada a **funcao2**, não existe um **break** ou um **return**, o processamento prosseguirá imediatamente abaixo, no código associado ao *label* [ **case 3 :** ].  
    Assim, imediatamente em seguida, chamará a **funcao3**, e só agora encontrará um **return**, que provocará retorno imediato.  
}
- c. ☐ se 'x' for **igual a 2**, a **funcao2** será chamada, em seguida a **funcao3** será chamada e em seguida 'x' será incrementada.
- {  
    **Errado.** Realmente **funcao2** e **funcao3** serão chamadas em sequência. Mas após executar **funcao3**, o processamento encontrará um desvio **return**, que provocará retorno imediato. Desse modo, a linha onde 'x' é incrementada nunca será atingida.  
}
- d. ☒ se 'x' for **igual a 2**, a **funcao2** será chamada, em seguida a **funcao3** será chamada, ocorrendo retorno em seguida.
- e. ☐ se 'x' for **igual a 3**, a **funcao3** será chamada e em seguida 'x' será incrementada.
- {  
    **Errado.** Há um **return** após a chamada a **funcao3**. Desse modo, a linha onde 'x' é incrementada nunca será atingida.  
}
- f. ☒ se 'x' for **igual a 3**, a **funcao3** será chamada, ocorrendo retorno em seguida.
- g. ☐ se 'x' for **igual a 4**, a **funcao4** será chamada e em seguida 'x' será

*incrementada.*

```
{
    Errado. Após a chamada a funcao4 não há nenhum desvio. Logo,
    antes de atingir a linha onde 'x' é incrementada, ocorrerá uma chama-
    da a funcao5.
}
```

h. ☒ se 'x' for **igual a 4**, a **funcao4** será chamada, em seguida a **funcao5** será chamada e em seguida 'x' será **incrementada**.

i. ☐ a **funcao5** será chamada **apenas quando 'x' for maior que 4**.

```
{
    Errado. A chamada à funcao5 está sob o label default.
    Logo, funcao5 será chamada se nenhum case acima tiver resultado
    verdadeiro, ou se o case imediatamente anterior tiver resultado ver-
    dadeiro, mas não contiver um desvio que impeça que o código de-
    fault seja executado.
    Então, se 'x' for maior que 4, apenas o default será executado e,
    realmente, funcao5 será chamada.
    Mas temos duas situações em que 'x' não é maior que 4, e a
    funcao5 será chamada:

```

- a. 'x' é menor que 1: não existem **cases** para essa situação; logo apenas o default será executado, e **funcao5** será chamada.
- b. 'x' é igual a 4: neste caso, será chamada a **funcao4** e, em segui- da, **também** será chamada a **funcao5**.

j. ☐ para qualquer valor **menor que 1 e maior que 4** a **funcao5** será cha- mada, ocorrendo retorno em seguida.

```
{
    Errado. Realmente funcao5 será chamada. Mas não há um return
    após essa chamada. Então, ao invés de retorno imediato, será atingi-
    da a linha onde 'x' é incrementada.
}
```

k. ☒ para qualquer valor menor que 1 e maior que 4 a **funcao5** será chama- da, e em seguida 'x' será **incrementada**.

{ A afirmação está **correta**. Lembrar apenas que, como exposto na resposta 'i', acima, caso 'x' seja **igual a 4**, a **funcao4** será chamada, e, em seguida, a **funcao5 também** será chamada, e finalmente 'x' será incrementada. }



**Para fazer:** crie um **projeto de teste** para as questões deste capítulo (quando isso **seja possível**). Procure **reproduzir, no código**, as situações descritas nas questões utilizadas. Introduza algumas **variações**.

## 6. Questões do Capítulo 6

Estas questões foram propostas na seção 6.5, página 206.

Cap.6 - 1. Assinale as afirmações corretas, considerando o código abaixo:

```
int x = 5;
int * px = &x;
```

- 
- a. ☐ *'px' é uma **referência** para 'x'.*  
    {  
        **Errado.**  
    }
- 
- b. ☒ *'px' é um **ponteiro** para 'x'.*
- 
- c. ☒ *'px' guarda o endereço de 'x'.*
- 
- d. ☐ *O valor armazenado em 'px' é 5.*  
    {  
        **Errado.**  
    }
- 
- e. ☒ *O valor armazenado em 'x' é 5.*
-

---

f. ☐ *É correto fazer: \*x = 10;*  
     {  
         **Errado.**  
     }

---

g. ☒ *É correto fazer: \*px = 10;*

---

**Cap.6 - 2.** Assinale as afirmações corretas, considerando o código abaixo:

```
int x = 5 ;
int & rx = x ;
```

---

a. ☒ 'rx' é uma **referência** para 'x'.

---

b. ☐ 'rx' é um **ponteiro** para 'x'.  
     {  
         **Errado.**  
     }

---

c. ☐ 'rx' *guarda o endereço de 'x'.*  
     {  
         **Errado.**  
     }

---

d. ☒ O valor armazenado em 'rx' é 5.

---

e. ☒ O valor armazenado em 'x' é 5.

---

f. ☒ 'rx' é um **sinônimo** de 'x'.

---

g. ☐ *É correto fazer: \*rx = 10;*  
     {  
         **Errado.**  
     }

---

h. ☒ *É correto fazer: x = 10;*

---

i. ☒ *É correto fazer: rx = 10;*

---

**Cap.6 - 3.** Assinale as afirmações corretas, considerando o código abaixo:

```
int x = 5 ;
int * px ;
px = &x ;
```

---

a. ☒ 'px' é uma **ponteiro** para 'x'.

---

b. ☒ O valor **apontado** por 'px', a partir da terceira linha, é 5.

---

c. ☐ *O código acima está incorreto e ocorrerá um erro de compilação.*  
     {  
         **Errado.**  
     }

---

**Cap.6 - 4.** Assinale as afirmações corretas, considerando o código abaixo:

```
int x = 5 ;
int & rx ;
rx = x ;
```

- 
- a. ☐ 'rx' é uma **referência** para 'x'.
- ```
{  
    Errado.  
}
```
- 
- b. ☐ O valor armazenado em 'rx' é 5.
- ```
{  
    Errado.  
}
```
- 
- c. ☒ O código acima está incorreto e ocorrerá um erro de compilação.
- 

---

## 7. Questões do Capítulo 7

---

Estas questões foram propostas na seção 7.6 página 260.

**Porque as estruturas (struct ou class) do C++ são melhores que as estruturas (struct) do C?**

- ☒ Uma struct ou class C++ permite:
- incluir funções como membros;
  - restringir o acesso a determinados membros de tal forma que só possam ser acessados nas funções declaradas dentro da própria struct ou class;
  - incluir uma função especial, a construtora, que irá garantir a correta inicialização dos membros;
  - por isso, a struct ou class C++ é melhor que a struct C: os dados serão protegidos pelas funções e assim poderemos garantir a sua inicialização e validação.

**O que são as restrições de acesso private e public ?**

- ☒ private restringe o acesso aos membros assim declarados; apenas as funções declaradas dentro da struct ou class poderão acessar os membros "private"; public libera o acesso aos membros assim declarados; qualquer função, seja ela membra da estrutura ou não, poderá acessar os membros "public".

**O que significa "this" ?**

- ☒ "this" é o parâmetro oculto recebido por funções membras de uma struct ou class; ele contém o endereço da variável (objeto) que foi utilizada para chamar a função; assim, neste exemplo,
- ```
Data dt ; dt.Imprime( );
```
- será chamada a função "Imprime" da class Data e o endereço de "dt" será recebido em "Imprime" no parâmetro "this".


**O que são funções operadoras ?**

- ☒ São funções criadas para serem utilizadas no formato "operador" e não apenas no formato "função". Isto é possível utilizando-se a palavra reservada "operator" mais um símbolo de operação para compor o nome da função.

**O que significa “const” na definição do tipo de um parâmetro de função?**

- ☒ “const “ indica que um ponteiro ou uma referência recebida como argumento, só poderá servir para que possamos ler o conteúdo apontado ou referenciado por esse ponteiro ou referência; não poderemos alterar esse conteúdo.

**Responda à pergunta que está no comentário do código abaixo:**

```
class X
{
    X( )  // Que função é esta e para que serve?
    {
    }
};
```


- ☒ Essa é a função construtora da “class X”, já que tem o mesmo nome da class. Serve para inicializar os membros de “X”.

**Responda à pergunta que está no comentário do código abaixo:**

```
class X
{
    ~X( )  // Que função é esta e para que serve?
    {
    }
};
```

- ☒ Essa é a função destrutora da class “X”, já que tem o mesmo nome da class precedido pelo sinal “~”.  
Serve para realizar qualquer finalização necessária (como fechar arquivos ou liberar memória alocada no heap), já que será chamada obrigatoriamente pelo menos uma vez: no momento em que um objeto dessa classe estiver para ser desalocado da memória.

**Responda às 4 perguntas que estão nos comentários do código abaixo:**

```
class X
{
    bool operator < ( const X & obj_2 ) const 
    // 1- Que função é esta e para que serve?
    // 2- Por que a palavra “const” aparece aí duas vezes?
    // 3- O que faz aí o “e-comercial” (&) ?
    // 4- O que faz aí a letra “X” ?
};
```

- ☒ 1 – Essa é uma função operadora que servirá sejam feitas operações relacionais “menor que” entre dois objetos dessa classe.  
Assim, esses dois objetos poderão ser comparados no formato operação: “ if ( x < y ); “
- ☒ 2 – O primeiro “const” especifica que o parâmetro “obj\_2” é uma referência “const” (seus membros poderão ser lidos, mas não alterados);  
O segundo “const” especifica que o ponteiro “this” é const (os membros apontados por ele poderão ser lidos mas não alterados).




- ☒ 3 – O “e-comercial” (&) indica que o parâmetro “obj\_2” está sendo passado por referência.
- ☒ 4- “X” é aí o tipo do parâmetro “obj\_2”; neste caso, então, “obj\_2” é um objeto da própria classe “X”.

## 8. Questões do Capítulo 11

Estas questões foram propostas na seção 11.4 página 315.


Responda à pergunta que está no comentário do código abaixo:

```
class X
{
};

class Y : public X   // O que significa esta declaração?
{
};
```

- ☒ Estamos declarando a class “Y” como uma derivada pública de “X”.

Responda à pergunta que está no comentário do código abaixo:

```
class X
{
    virtual bool Funcao( )   // Que função é esta
                             e para que serve?
    {
    }
};
```

- ☒ Essa é uma função virtual.  
Sendo assim, servirá para que uma derivada possa substituí-la (declarando uma função de mesmo nome).

Essa substituição ocorrerá mesmo que “Funcao” seja chamada por outra função qualquer da “class X”.

Pois, sendo virtual, será chamada por endereço (armazenado na “virtual table”).


Assim, havendo uma derivada e tendo sido criado um objeto dessa derivada, esse endereço será alterado antes que a construtora da derivada seja executada.

Desse modo, ao invés do endereço de “Funcao” da base, a “virtual table” conterá o endereço de “Funcao” da derivada.

Responda à pergunta que está no comentário do código abaixo:

```
class X
{
    X ( )
    {
    }
};
```

```


class Y : public X
{
    Y ( )
    {
    }
};
int main ( )
{
    Y meu_Y ;  // Que funções serão executadas aqui ?
                E em que ORDEM ?
}

```

- ☒ a) será chamada a construtora “Y”;
- b) antes que a construtora “Y” seja executada, será chamada e executada a construtora “X”;
- c) finalmente será executada a construtora “Y”.

responda à pergunta que está no comentário do código abaixo:

```

class X
{
    int Imprime ( )
    {
        // ....
    }
};
class Y : public X
{
    int Imprime( )
    {
        //...
    }
};
int main ( )
{
    Y meu_Y ;
    meu_Y.Imprime( ) ;  // Que funções (e de quais classes)
                        // serão chamadas aqui ?
                        // E em que ORDEM ?
    // ...
}

```

- ☒ Será executada apenas: Y::Imprime( )  
(função “Imprime” da “class Y”).


Responda à pergunta que está no comentário do código abaixo:

```

class X
{
    virtual bool Funcao( ) {
        // ...
    }
    int Imprime ( ) {
        Funcao( ) ;
        // ...
    }
}

```

```

};
class Y : public X
{
    bool Funcao( ) {
        // ...
    }
};
int main ( )
{
    Y meu_Y ;
    meu_Y.Imprime( ) ;  // Que funções (e de quais classes)
                        // serão chamadas aqui ?
                        // E em que ORDEM ?
    // ...
}

```

- ☒ a) será executada X::Imprime( )  
(função “Imprime” da “class X”)
- b) será executada Y::Funcao()  
(função “Funcao” da “class Y”)

**Responda à pergunta que está no comentário do código abaixo:**

```

template <class T> class X
{
    T Valor ;  // Qual é o tipo do membro de dados “Valor” ?
};

```

- ☒ Como “X” é um template de class, “T” é, por enquanto, apenas uma parametrização de tipo e, assim, “Valor” não tem ainda um tipo definido.

Assim sendo, quando criarmos um objeto usando esse template, obrigatoriamente teremos que indicar um tipo.

Então, “Valor” será criado com o tipo que indicarmos na **declaração de cada objeto**.

Exemplos:

X < int > x1; // Valor será do tipo int.

X <double > x2; // Valor será do tipo double.

---

## Anexo C. Guia de consulta rápida

---

|                                                                  |     |
|------------------------------------------------------------------|-----|
| 1. Tabela de tipos primitivos.....                               | 477 |
| 2. Tabela de operadores <i>básicos</i> por tipo de operação..... | 479 |
| 3. Tabela completa de operadores e suas precedências.....        | 481 |
| 4. Controles de fluxo de processamento.....                      | 483 |
| 5. Principais diretivas de compilação.....                       | 485 |
| 6. sequências <i>escape</i> .....                                | 486 |
| 7. Palavras reservadas.....                                      | 486 |
| 8. Modificadores.....                                            | 487 |

## 1. Tabela de tipos primitivos

| Tipo                          | Tamanho mínimo (*1) em bytes                                                                                                                                                                | Intervalo de valores                                                                                                                            | Forma de armazenamento / natureza                                                  |
|-------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------|
| <b>unsigned char</b>          | 1                                                                                                                                                                                           | 0 a 255                                                                                                                                         | inteiro                                                                            |
| <b>char</b>                   | 1                                                                                                                                                                                           | -128 a 127                                                                                                                                      | inteiro                                                                            |
| <b>unsigned short</b>         | 2                                                                                                                                                                                           | 0 a 65535                                                                                                                                       | inteiro                                                                            |
| <b>short</b>                  | 2                                                                                                                                                                                           | -32768 a 32767                                                                                                                                  | inteiro                                                                            |
| <b>unsigned long</b>          | 4                                                                                                                                                                                           | 0 a +4294967295                                                                                                                                 | inteiro                                                                            |
| <b>long</b>                   | 4                                                                                                                                                                                           | -2147483648 a + 2147483647                                                                                                                      | inteiro                                                                            |
| <b>unsigned int</b>           | <b>word (*2)</b>                                                                                                                                                                            | <b>(*2)</b>                                                                                                                                     | inteiro                                                                            |
| <b>int</b>                    | <b>word (*2)</b>                                                                                                                                                                            | <b>(*2)</b>                                                                                                                                     | inteiro                                                                            |
| <b>unsigned long long</b>     | 64 bits sem sinal                                                                                                                                                                           |                                                                                                                                                 | inteiro                                                                            |
| <b>long long</b>              | 64 bits com sinal                                                                                                                                                                           |                                                                                                                                                 | inteiro                                                                            |
| <b>size_t</b>                 | - <b>pode</b> ser equivalente ao <b>unsigned int</b> (em plataformas de <b>32 bits</b> );<br>- ou <b>pode</b> ser equivalente ao <b>unsigned long long</b> (plataformas de <b>64 bits</b> ) |                                                                                                                                                 | <b>inteiro</b> (usado para <b>dimensões</b> em geral e <b>índices</b> de vetores). |
| <b>enum</b>                   | De <b>1 byte</b> até, no máximo, o tamanho que tenha o <b>int</b> , dependendo dos <b>valores usados</b> .                                                                                  |                                                                                                                                                 | <b>inteiro</b> (enumera <b>constantes</b> )                                        |
| <b>enum : &lt;tipo&gt;</b>    | Assume o indicado em <b>&lt;tipo&gt;</b> , que deve ser um <b>inteiro (C++11)</b>                                                                                                           |                                                                                                                                                 | o mesmo que o anterior                                                             |
| <b>bool</b>                   | 1                                                                                                                                                                                           | true / false (1 ou zero)                                                                                                                        | inteiro (só <b>C++</b> )                                                           |
| <b>float</b>                  | 4                                                                                                                                                                                           | 3.4E-38 a 3.4E+38                                                                                                                               | ponto flutuante                                                                    |
| <b>double</b>                 | 8                                                                                                                                                                                           | 1.7E-308 a 1.7E+308                                                                                                                             | ponto flutuante                                                                    |
| <b>long double</b>            | 10 (*4)                                                                                                                                                                                     | 3.4E-4932 a 1.1E+4932                                                                                                                           | ponto flutuante                                                                    |
| <b>ponteiro &lt;tipo&gt;*</b> | <b>word (*2)</b>                                                                                                                                                                            | <b>(*2)</b>                                                                                                                                     | endereço (*3)                                                                      |
| <b>void *</b>                 | <b>word (*2)</b>                                                                                                                                                                            | <b>Caso especial para ponteiros:</b><br>um ponteiro que não conhece o tipo apontado (e precisará sofrer um <b>cast</b> , antes que seja usado). | endereço (*3)                                                                      |
| <b>void</b>                   | <b>Vazio:</b> não é possível declarar objetos desse tipo. Usado apenas para definir o retorno de funções que não retornam valor (ausência de retorno).                                      |                                                                                                                                                 |                                                                                    |

|      |                                                                                                                                                                                                                                             |
|------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| (*1) | O que o padrão de <b>C++</b> garante são os tamanhos <b>mínimos</b> para cada tipo. Poderão ocupar mais <i>bytes</i> , dependendo da plataforma. Mas, para saber se um tipo é capaz de armazenar um determinado valor, independentemente da |
|------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

|      |                                                                                                                                                                                                                                                                                                         |
|------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|      | plataforma, precisamos nos basear na <b>garantia mínima</b> . Além disso, observar que o tamanho do tipo " <b>int</b> " <b>sempre</b> depende da plataforma (ver a nota *2). E o tipo " <b>long double</b> " pode não ser suportado (ver a nota *4).                                                    |
| (*2) | <b>Word</b> (" <i>palavra da máquina</i> "): o <b>int</b> , como os <b>ponteiros</b> , têm o <b>tamanho da palavra da máquina</b> . Exemplos: em sistemas de 16 bits, o <b>int</b> tem <b>2 bytes</b> ; em sistemas de 32 bits, tem <b>4 bytes</b> ; em sistemas de 64 bits, pode ter <b>8 bytes</b> .. |
| (*3) | Um <b>ponteiro</b> é um número <b>inteiro sem sinal</b> que armazena <b>endereços de memória</b> e não valores.                                                                                                                                                                                         |
| (*4) | O tipo <b>long double</b> não é suportado em vários sistemas, sendo substituído por <b>double</b> .                                                                                                                                                                                                     |

## 2. Tabela de operadores *básicos* por tipo de operação

### OPERADORES ARITMÉTICOS

|    |                                              |
|----|----------------------------------------------|
| -  | Subtração                                    |
| +  | Adição                                       |
| *  | Multiplicação                                |
| /  | Divisão                                      |
| %  | Resto de uma divisão de inteiros (módulos)   |
| -- | Decrementa em 1 (pode ser pré ou pós-fixado) |
| ++ | Incrementa em 1 (pode ser pré ou pós-fixado) |

### OPERADORES RELACIONAIS

|    |                               |
|----|-------------------------------|
| >  | Maior que                     |
| >= | Maior que ou igual            |
| <  | Menor que                     |
| <= | Menor que ou igual            |
| == | Igual ( <i>equal</i> )        |
| != | Desigual ( <i>not equal</i> ) |

### OPERADORES LÓGICOS

|    |                    |
|----|--------------------|
| && | E ( <b>AND</b> )   |
|    | Ou ( <b>OR</b> )   |
| !  | Não ( <b>NOT</b> ) |

### OPERADORES BIT A BIT

|    |                                                          |
|----|----------------------------------------------------------|
| &  | E ( <b>AND</b> )                                         |
|    | Ou ( <b>OR</b> )                                         |
| ^  | Ou Exclusivo ( <i>Exclusive Or</i> ou <b>XOR</b> )       |
| ~  | Negação para bits ( <b>NOT</b> ) - ou "complemento de 1" |
| >> | Deslocamento à direita                                   |
| << | Deslocamento à esquerda                                  |

### OUTROS OPERADORES

|        |              |                                                                                                                                                                                                   |
|--------|--------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ?      | :            | Sintaxe: <b>condição ? expressão1 : expressão2</b> .<br>Retorno: se a <b>condição</b> for verdadeira retorna a avaliação da <b>expressão1</b> , caso contrário a avaliação da <b>expressão2</b> . |
| &      |              | <b>&amp;xxx</b> retorna o endereço de <b>xxx</b> (ponteiro a <b>xxx</b> ).                                                                                                                        |
| *      |              | <b>*yyy</b> retorna o valor do objeto cujo endereço está em <b>yyy</b> .                                                                                                                          |
| sizeof | (<x>)        | Retorna o tamanho, em bytes, de <b>x</b> (tipo, objeto ou ponteiro)                                                                                                                               |
| ,      |              | sequencia de diversas operações separadas pela vírgula.                                                                                                                                           |
| .      |              | Referencia os elementos de uma <b>struct</b> ou de uma <b>union</b> .                                                                                                                             |
| ->     |              | Ponteiro aos elementos de uma <b>struct</b> ou de uma <b>union</b>                                                                                                                                |
| (      | <operação> ) | Aumenta a precedência da operação dentro dos parênteses                                                                                                                                           |
| (tipo) | <d>          | Força a modificação de tipo ( <b>cast</b> ) de dados ( <b>*1</b> )                                                                                                                                |
| [      | ]            | Dimensão e índice de uma matriz.                                                                                                                                                                  |
| <f>    | ()           | Os parênteses representam aí uma <b>chamada à função</b> " <b>f</b> "                                                                                                                             |

**OPERADORES COMBINADOS**

(uma operação aritmética ou bit-a-bit, seguida de uma operação de atribuição)

**Operador    Forma equivalente****x += y**     $x = x + y$ **x -= y**     $x = x - y$ **x \*= y**     $x = x * y$ **x /= y**     $x = x / y$ **x %= y**     $x = x \% y$ **x >>= y**     $x = x >> y$ **x <<= y**     $x = x << y$ **x &= y**     $x = x \& y$ **x ^= y**     $x = x \wedge y$ **x |= y**     $x = x | y$ **(\*1) - CASTS - o operador de molde ou conversão de tipo.**

O operador de molde converte uma expressão para um determinado tipo.

**Exemplos:**`int r = (int) (pi * raio);    // o resultado da multiplicação é convertida para int``float a = (float)b / 3;    // 'b' é convertido para float e em seguida é dividido por 3`Em **C++** (mas não em **C89**) podemos usar esse operador no “**formato função**”:`int r = int ( pi * raio );    // o resultado da multiplicação é convertida para int``float a = float ( b ) / 3 ;    // 'b' é convertido para float e em seguida é dividido por 3`



### 3. Tabela completa de operadores e suas precedências

**Cada grupo tem precedência maior que o grupo seguinte.  
Dentro do mesmo grupo todos têm a mesma precedência.**

| Operadores                                                                                                                                                                                                                                                                         | Descrição                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          | Associati-<br>vidade               |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------|
| <code>::</code>                                                                                                                                                                                                                                                                    | Resolução de escopo                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                | ➔<br>esquerda<br>para direi-<br>ta |
| <code>++</code><br><code>--</code><br><code>( )</code><br><code>[ ]</code><br><code>.</code><br><code>-&gt;</code><br><code>typeid(&lt;t&gt;)</code><br><code>const_cast</code><br><code>dynamic_cast</code><br><br><code>static_cast</code><br><br><code>reinterpret_cast</code>  | Incremento pós-fixado<br>Decremento pós-fixado<br>Parênteses (agrupamento de operações)<br>Índice de vetor<br>Ligação de membro a objeto<br>Ligação de membro a ponteiro para objeto<br>Informações sobre um tipo em <i>run-time</i> .<br>Conversão de <i>const</i> para <i>não-const</i><br>Conversão de tipos relacionáveis com verifica-<br>ção em tempo de execução<br>Conversão de tipos relacionáveis sem verifica-<br>ção<br>Conversão de tipos não relacionáveis                                           | ➔<br>esquerda<br>para direi-<br>ta |
| <code>++</code><br><code>--</code><br><code>+</code><br><code>-</code><br><code>!</code><br><code>~</code><br><code>*</code><br><code>&amp;</code><br><br><code>new</code><br><code>new[]</code><br><code>delete</code><br><code>delete[]</code><br><code>sizeof(&lt;t&gt;)</code> | Incremento pré-fixado<br>Decremento pré-fixado<br>Adição unária<br>Subtração unária<br><i>Not</i> lógico (negação)<br><i>Not</i> para <i>bits</i> (ou complemento)<br><i>Pointer</i> (Ponteiro, de-referência)<br><i>Address Of</i> (retorna endereço de um objeto)<br>Alocação dinâmica de memória ( <b>1</b> elemento).<br>Alocação dinâmica de memória ( <b>n</b> elementos).<br>Libera memória alocada por <b>new</b> .<br>Libera memória alocada por <b>new[]</b> .<br>Tamanho de um tipo, objeto ou ponteiro | ←<br>direita<br>para es-<br>querda |
| <code>(tipo) d e</code><br><code>tipo(d)</code>                                                                                                                                                                                                                                    | Conversão ( <i>cast</i> ) genérica de tipo de dados                                                                                                                                                                                                                                                                                                                                                                                                                                                                |                                    |
| <code>.*</code><br><code>-&gt;*</code>                                                                                                                                                                                                                                             | Ligação de objeto a ponteiro para membro.<br>Ligação de ponteiro para objeto a ponteiro<br>para membro                                                                                                                                                                                                                                                                                                                                                                                                             | ➔<br>esquerda<br>para direi-       |

|                                                                 |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |                                    |
|-----------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------|
| *<br>/<br>%                                                     | Multiplicação<br>Divisão<br>Módulo (resto)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | ta                                 |
| +<br>-                                                          | Adição<br>Subtração                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |                                    |
| <<<br>>>                                                        | Deslocamento de bits à esquerda<br>Deslocamento de bits à esquerda à direita                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |                                    |
| <<br><=<br>><br>>=                                              | "menor que"<br>"menor ou igual que"<br>"maior que"<br>"maior ou igual que"                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |                                    |
| ==<br>!=                                                        | "Igual a"<br>"diferente de "                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |                                    |
| &                                                               | <i>And</i> para bits                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |                                    |
| ^                                                               | <i>Or exclusivo</i> para bits                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                    |
|                                                                 | <i>Or</i> para bits                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |                                    |
| &&                                                              | <i>And</i> lógico                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |                                    |
|                                                                 | <i>Or</i> lógico                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |                                    |
| <c> ? <t> : <f>                                                 | Condicional ternário (condição) ? <i>true</i> : <i>false</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |                                    |
| =<br>+=<br>-=<br>*=<br>/=<br>%=<br><<=<br>>>=<br>&=<br> =<br>^= | Atribuição<br>x += y : Soma 'x' e 'y', atribui resultado a 'x'<br>x -= y : Subtrai 'y' de 'x', atribui resultado a 'x'<br>x *= y : Multiplica 'x' por 'y', atribui resultado a 'x'<br>x /= y : Divide 'x' por 'y', atribui resultado a 'x'<br>x %= y : Resto de 'x' dividido por 'y'; atribui a 'x'<br>x <<= y : Desloca(←) 'y' <i>bits</i> em 'x'; atribui a 'x'<br>x >>= y : Desloca(→) 'y' <i>bits</i> em 'x'; atribui a 'x'<br>x &= y : <i>And</i> de <i>bits</i> (x & y); atribui a 'x'<br>x  = y : <i>Or</i> de <i>bits</i> (x   y); atribui a 'x'<br>x ^= y : <i>Xor</i> de <i>bits</i> (x ^ y); atribui a 'x' | ←<br>Direita<br>para es-<br>querda |
| throw                                                           | Lançamento de exceção                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 | -x-                                |
| ,                                                               | Vírgula: sequencia de diversas operações se-<br>paradas pela vírgula.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 | →<br>esquerda<br>para direi-<br>ta |

## 4. Controles de fluxo de processamento

|                                                                                                      |                                                                                                                                                                                                                                                                                                                            |
|------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [ função( ) ]                                                                                        | { }                                                                                                                                                                                                                                                                                                                        |
| {                                                                                                    | Abertura de um bloco de instruções, esteja ele iniciando uma função ou mesmo um grupo qualquer de instruções dentro da função (bloco aninhado)                                                                                                                                                                             |
|                                                                                                      | [ declaração de variáveis ou constantes ; ]                                                                                                                                                                                                                                                                                |
|                                                                                                      | ...                                                                                                                                                                                                                                                                                                                        |
|                                                                                                      | instrução ;                                                                                                                                                                                                                                                                                                                |
|                                                                                                      | ...                                                                                                                                                                                                                                                                                                                        |
| }                                                                                                    | Fechamento de um bloco de instruções, esteja ele encerrando uma função ou mesmo um grupo qualquer de instruções dentro da função (bloco aninhado)                                                                                                                                                                          |
|                                                                                                      | Um bloco pode ser o corpo completo de uma função ou um bloco interno (aninhado) à uma função.<br>Aloca memória para a <b>declaração</b> (quanto presente) e executa cada <b>instrução</b> sequencialmente, a não ser que o controle de fluxo force uma transferência para outra instrução, desviando da sequência natural. |
| if (expressão) [ { } <instrução; instrução; ...> [ } ] [else [ { } <instrução; instrução;...>[ } ] ] |                                                                                                                                                                                                                                                                                                                            |
|                                                                                                      | Se a expressão for verdadeira, o primeiro bloco de instruções será executado. Caso contrário o segundo bloco (se existir) será executado.<br>Se houver mais do que uma instrução, os blocos de instruções associados ao <b>if</b> ou ao <b>else</b> devem iniciar e encerrar com chaves [ { ...; ....; .... } ].           |
| switch (expressão)                                                                                   |                                                                                                                                                                                                                                                                                                                            |
|                                                                                                      | {                                                                                                                                                                                                                                                                                                                          |
|                                                                                                      | [ declaração ; ]                                                                                                                                                                                                                                                                                                           |
|                                                                                                      | case <constante 1> :                                                                                                                                                                                                                                                                                                       |
|                                                                                                      | instrução;                                                                                                                                                                                                                                                                                                                 |
|                                                                                                      | ...                                                                                                                                                                                                                                                                                                                        |
|                                                                                                      | [ case <constante N> :                                                                                                                                                                                                                                                                                                     |
|                                                                                                      | instrução ; ]                                                                                                                                                                                                                                                                                                              |
|                                                                                                      | ....                                                                                                                                                                                                                                                                                                                       |
|                                                                                                      | [ default :                                                                                                                                                                                                                                                                                                                |
|                                                                                                      | instrução ; ]                                                                                                                                                                                                                                                                                                              |
|                                                                                                      | ...                                                                                                                                                                                                                                                                                                                        |
|                                                                                                      | }                                                                                                                                                                                                                                                                                                                          |
|                                                                                                      | Avalia <b>expressão</b> e executa as instruções associadas à constante (" <b>case</b> ") que tenha este valor. Caso nenhuma constante preencha esta condição, executa o <b>statement default</b> . As declarações são locais e não permitem inicialização.                                                                 |

|                                                                                                     |                                                                                                                                                                                                                                                                                       |
|-----------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>for ( [início] ; [condição] ; [ progressão ] ) [ { } [ &lt;instrução&gt; ] ; [ ... ; ] [ } ]</b> |                                                                                                                                                                                                                                                                                       |
|                                                                                                     | Define um laço. Executa a instrução enquanto “ <b>condição</b> ” for verdadeira; se houver mais do que uma instrução, o bloco de instruções associado ao <b>for</b> , deve iniciar e encerrar com chaves [ { ... ; ... ; } ].                                                         |
| <b>início</b>                                                                                       | Zero ou mais expressões separadas por vírgulas que serão executadas antes do laço.                                                                                                                                                                                                    |
| <b>condição</b>                                                                                     | O laço será executado enquanto esta condição for verdadeira.                                                                                                                                                                                                                          |
| <b>progressão</b>                                                                                   | Zero ou mais expressões, separadas por vírgulas, que serão avaliadas após cada passo.                                                                                                                                                                                                 |
| <b>while (expressão) [ { } [ &lt;instrução&gt; ] ; [ ... ; ] [ } ]</b>                              |                                                                                                                                                                                                                                                                                       |
|                                                                                                     | Define um laço. Executa a instrução enquanto “ <b>condição</b> ” for verdadeira; se houver mais do que uma instrução, o bloco de instruções associado ao <b>for</b> , deve iniciar e encerrar com chaves [ { ... ; ... ; ... } ].                                                     |
| <b>do [ { } &lt;instruções&gt; ; [ } ] while (expressão) ;</b>                                      |                                                                                                                                                                                                                                                                                       |
|                                                                                                     | Define um laço que será executado, no mínimo uma vez. A partir daí continuará executando enquanto a expressão for verdadeira. Se houver mais do que uma instrução, o bloco de instruções associado ao <b>do...while</b> , deve iniciar e encerrar com chaves [ { ... ; ... ; ... } ]. |
| <b>break ;</b>                                                                                      |                                                                                                                                                                                                                                                                                       |
|                                                                                                     | Finaliza os laços <b>for</b> , <b>while</b> ou <b>do...while</b> mais recentes; ou o <b>switch</b> mais recente.                                                                                                                                                                      |
| <b>continue ;</b>                                                                                   |                                                                                                                                                                                                                                                                                       |
|                                                                                                     | Salta as instruções posteriores em um laço, transferindo o fluxo para a próxima avaliação de condição.                                                                                                                                                                                |
| <b>goto &lt;rótulo&gt; ;</b>                                                                        |                                                                                                                                                                                                                                                                                       |
|                                                                                                     | Desvia o fluxo do programa para o rótulo especificado. O rótulo pode anteceder qualquer instrução da função e deve ser seguido por “:”. Um rótulo obedece ao escopo da função; isto é, não é possível executar <b>goto</b> para qualquer rótulo que não esteja na mesma função.       |
| <b>return [expressão] ;</b>                                                                         |                                                                                                                                                                                                                                                                                       |
|                                                                                                     | Finaliza uma função e opcionalmente retorna o valor da expressão (no caso de funções cujo tipo de retorno não seja <b>void</b> ).                                                                                                                                                     |

## 5. Principais diretivas de compilação

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b><code>/* &lt;comentário&gt; */</code> e <code>// &lt;comentário&gt;</code></b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| Identificam comentários – o compilador ignora tudo o que estiver entre <code>/*</code> e <code>*/</code> , ou após o <code>//</code> até o final da linha de texto.                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <b><code>#define &lt;identificador&gt; &lt;substituto&gt;</code></b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| Substitui as ocorrências do identificador pelo substituto (simples troca).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <b><code>#undef &lt;identificador&gt;</code></b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| Cancela a definição anterior de identificador criado por <b><code>#define</code></b> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b><code>#if &lt;expressão&gt;</code></b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| Permite compilação condicional das declarações seguintes se o valor da <b><code>expressão</code></b> for verdadeiro.                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <b><code>#else</code></b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| Estabelece uma alternativa se a condição <b><code>#if</code></b> fracassar.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b><code>#endif</code></b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| Usada para identificar o fim de uma <b>compilação condicional</b> iniciada com <b><code>#if</code></b> , <b><code>#ifdef</code></b> ou <b><code>#ifndef</code></b> .                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <b><code>#ifdef &lt;identificador&gt;</code> ou <code>#if defined &lt;identificador&gt;</code></b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| Se o <b>identificador</b> foi <b>definido</b> anteriormente (com <b><code>#define</code></b> ), as instruções até <b><code>#else</code></b> (se existir) ou até <b><code>#endif</code></b> são compiladas.                                                                                                                                                                                                                                                                                                                                                                                     |
| <b><code>#ifndef &lt;identificador&gt;</code></b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| Se o <b>identificador</b> <b>não foi definido</b> anteriormente as instruções até <b><code>#else</code></b> (se existir) ou até <b><code>#endif</code></b> são compiladas.                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <p><b>Exemplo com <code>#ifdef ... #else ... #endif</code> :</b> <i>// se o identificador “WINDOWS” <code>#ifndef WINDOWS</code> // foi anteriormente criado com um <code>#define</code></i></p> <p><i>// ... código específico para Windows ...</i></p> <p><b><code>#else</code></b></p> <p><i>// ... código específico para outras plataformas (Unix, por exemplo) ...</i></p> <p><b><code>#endif</code></b></p> <p>Apenas uma dessas duas seções de código será levada em conta pelo compilado (dependendo da existência ou não do identificador “WINDOWS”). A outra será desprezada. .</p> |
| <b><code>#error &lt;mensagem&gt;</code></b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Na compilação, mostra a <code>&lt;mensagem&gt;</code> e encerra a compilação.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b><code>#include &lt;arquivo&gt;</code> ou <code>#include “arquivo”</code></b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <p>Inclui o conteúdo do arquivo especificado no módulo corrente.</p> <p>O nome do arquivo pode estar entre <b>aspas</b> ou entre um sinal de <b>menor</b> [<code>&lt;</code>] e um sinal de <b>maior</b> [<code>&gt;</code>], o que estabelece a seguinte diferença:</p> <p><b>“arquivo”</b> : Busca o arquivo especificado no diretório corrente.</p> <p><b>&lt;arquivo&gt;</b> : Não considera o diretório corrente e busca o arquivo em diretórios definidos por variáveis de ambientes ou por argumentos passados para o compilador.</p>                                                   |

## 6. seqüências escape

| Código | Valor | ASCII | Significado                                                                                                              |
|--------|-------|-------|--------------------------------------------------------------------------------------------------------------------------|
| \a     | 0x07  | BEL   | Campainha                                                                                                                |
| \b     | 0x08  | BS    | Retrocesso ( <i><u>B</u>ackspace</i> )                                                                                   |
| \f     | 0x0C  | FF    | Quebra de Folha ( <i><u>F</u>orm feed</i> )                                                                              |
| \n     | 0x0A  | LF    | Quebra de Linha ( <i><u>N</u>ewline ou <u>L</u>ine feed</i> )                                                            |
| \r     | 0x0D  | CR    | Retorno de Carro ( <i><u>C</u>arriage <u>R</u>eturn</i> )                                                                |
| \t     | 0x09  | HT    | Tabulação ( <i><u>T</u>abulação (Horizontal)</i> )                                                                       |
| \v     | 0x0B  | VT    | Tabulação ( <i><u>V</u>ertical</i> )                                                                                     |
| \\     | 0x5C  | \     | Barra invertida (literal). <i>Exemplo:</i><br><b>cout &lt;&lt; "\\";</b> - <i>imprime a segunda barra.</i>               |
| \'     | 0x2C  | '     | Apóstrofe (literal): '\'; é considerado o caractere '                                                                    |
| \"     | 0x22  | "     | Aspas (literal): "\""; é considerado o caractere "                                                                       |
| \ooo   |       |       | Qualquer número em octal. <i>Exemplo:</i><br><b>cout &lt;&lt; "\101";</b> - <i>imprime A (decimal 65 em ASCII)</i>       |
| \xhhh  |       |       | Qualquer número em hexadecimal. <i>Exemplo:</i><br><b>cout &lt;&lt; "\x41";</b> - <i>imprime A (decimal 65 em ASCII)</i> |

## 7. Palavras reservadas

|           |           |          |           |
|-----------|-----------|----------|-----------|
| asm       | extern    | return   | class     |
| auto      | far       | short    | public    |
| break     | float     | signed   | private   |
| case      | for       | sizeof   | protected |
| cdecl     | goto      | static   | friend    |
| char      | huge      | struct   | inline    |
| const     | if        | switch   | this      |
| continue  | int       | typedef  | delete    |
| default   | interrupt | union    | new       |
| do        | long      | unsigned | operator  |
| double    | near      | void     | virtual   |
| else      | pascal    | volatile | try       |
| enum      | register  | while    | catch     |
| namespace | mutable   | explicit | throw     |
| using     | template  | typeid   | typename  |
| true      | false     | bool     |           |



Atenção: palavras reservadas não podem ser usadas como nomes de identificadores.

## 8. Modificadores

### MODIFICADORES DE TIPO

|                 |                                                                    |
|-----------------|--------------------------------------------------------------------|
| <b>signed</b>   | Número com sinal.                                                  |
| <b>unsigned</b> | Número sem sinal.                                                  |
| <b>long</b>     | Número inteiro com o tamanho mínimo garantido de <b>4 bytes</b> .  |
| <b>short</b>    | Número inteiro com o tamanho mínimo garantido de <b>2 bytes</b> .. |

### MODIFICADORES DE ACESSO

|                 |                                                                                 |
|-----------------|---------------------------------------------------------------------------------|
| <b>const</b>    | Valor constante; inicialização obrigatória; não alterável ( <b>read-only</b> ). |
| <b>volatile</b> | Valor pode <b>mudar a qualquer momento</b> (não deve ser otimizado).            |

### MODIFICADORES DE CLASSE DE ARMAZENAMENTO

|                 |                                                                                                                                                                                                                                                                                                |
|-----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>auto</b>     | Não reserva espaço permanente de memória. Variáveis desta classe são alocadas na pilha, sendo <b>desalocadas automaticamente, ao final do bloco de sua declaração</b> . É a classe <i>default</i> , para variáveis declaradas dentro de blocos.                                                |
| <b>extern</b>   | Indica ao compilador que as variáveis que o seguem serão declarados em algum outro módulo. Usado quando mais de um arquivo compartilha as mesmas variáveis <b>globais</b> , as quais reservam <b>memória permanente (tempo de vida global)</b> .                                               |
| <b>static</b>   | Em um <b>escopo local</b> , define variáveis locais ao bloco, mas que <b>manterão seus valores</b> após o retorno da função. No <b>escopo global</b> , variáveis com <b>tempo de vida global (permanentes)</b> , mas <b>visibilidade restrita ao módulo</b> (arquivo) em que foram declaradas. |
| <b>register</b> | Sugere ao compilador manter o valor da variável em <b>registrador</b> da CPU ao invés da pilha (uma "dica" de otimização, desnecessária em compiladores modernos).                                                                                                                             |

### MODIFICADORES DE MEMÓRIA (não padrão; para plataformas de 16 bits)

|                            |                                                                                                                         |
|----------------------------|-------------------------------------------------------------------------------------------------------------------------|
| <b>&lt;tipo&gt; near *</b> | Ponteiros que armazenam apenas o <i>offset</i> , que representa um deslocamento dentro de um mesmo segmento de memória. |
| <b>&lt;tipo&gt; far *</b>  | Ponteiros que armazenam o segmento e o <i>offset</i> .                                                                  |

### MODIFICADORES DA FORMA DE CHAMADA DE FUNÇÕES



Não podem ser usados para funções membras de classe, exceto funções estáticas, que não recebem o *this* como argumento implícito.

|               |                                                                                                                                                                                                                                                                                                                                                                                                            |
|---------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>pascal</b> | Usa convenção de passagem de parâmetros ao modo de Pascal. Os parâmetros são colocados na pilha da esquerda para a direita e é responsabilidade da função chamada limpar a pilha antes de retornar.                                                                                                                                                                                                        |
| <b>cdecl</b>  | Oposto de <b>pascal</b> . Os parâmetros são colocados na pilha da direita para a esquerda e é responsabilidade de quem chama limpar a pilha após o retorno da função chamada. Em <b>C++</b> , <b>na maioria dos compiladores</b> , esse é o <i>default</i> para funções globais e estáticas membras de classe. Em <b>C</b> , para funções globais e estáticas de módulo (sempre dependendo do compilador). |