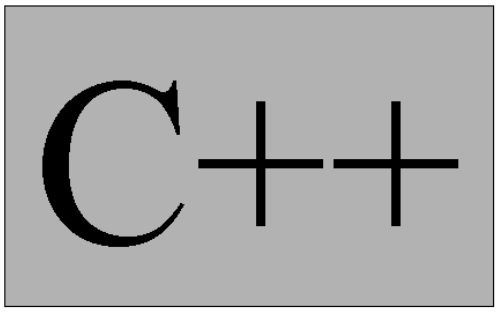
PROGRAMAÇÃO ORIENTADA A OBJETOS EM



BERILHES BORGES GARCIA
FLÁVIO MIGUEL VAREJÃO

COLABORAÇÃO: FABRÍCIA DOS SANTOS NASCIMENTO

ÍNDICE

1.1 ABSTRAÇÃO DE DADOS 3

1.2 HISTÓRICO 4

1.3 A LINGUAGEM C++ 5

2.1 INTRODUÇÃO 7

2.2.1 PROBLEMA EXEMPLO 8

2.2 UM EXEMPLO DE ABSTRAÇÃO DE DADOS EM C++ 8

2.2.1 ESPECIFICAÇÃO E IMPLEMENTAÇÃO 9

2.3 UM EXEMPLO DE ESPECIFICAÇÃO 10

2.3.1 ESPECIFICAÇÃO DA CLASSE BIGINT 10
2.3.2 OCULTAMENTO DE INFORMAÇÕES 12
2.3.3 FUNÇÕES MEMBROS 13
2.3.4 OVERLOADING 13
2.3.5 FUNÇÕES COM ARGUMENTOS DEFAULT 14
2.3.6 CHAMANDO FUNÇÕES MEMBROS 15
2.3.7 CONSTRUTORES 15
2.3.8 CONSTRUTORES E CONVERSÃO DE TIPOS 16
2.3.9 CONSTRUTORES E INICIALIZAÇÃO 17
2.3.10 OVERLOADING DE OPERADORES 18
2.3.11 DESTRUTORES 19

3.1 INTRODUÇÃO 20

3.2 IMPLEMENTAÇÃO DE UMA CLASSE BIGINT 21

3.2.1 O CONSTRUTOR BIGINT(CONST CHAR*) 21
3.2.2 O OPERADOR RESOLUÇÃO DO ESCOPO 22
3.2.3 CONSTANTES 22
3.2.4 FUNÇÕES MEMBROS CONSTANTES 23
3.2.5 O OPERADOR NEW 24
3.2.6 DECLARAÇÃO EM BLOCOS 24
3.2.7 O CONSTRUTOR BIGINT (UNSIGNED) 25
3.2.8 O CONSTRUTOR CÓPIA 25
3.2.9 REFERÊNCIAS 26
3.2.10 O OPERADOR ADIÇÃO DA CLASSE BIGINT 28
3.2.11 FUNÇÕES FRIEND 31
3.2.12 A PALAVRA RESERVADA THIS 32
3.2.13 O OPERADOR ATRIBUIÇÃO DA CLASSE BIGINT 32
3.2.14 A FUNÇÃO MEMBRO BIGINT::PRINT() 33
3.2.15 O DESTRUTOR DA CLASSE BIGINT 33
3.2.16 FUNÇÕES INLINE 34

4.1 INTRODUÇÃO 36

4.1.1 CLASSES DERIVADAS 37
4.1.2 REDECLARAÇÃO - FUNÇÕES MEMBROS VIRTUAIS 39

4.2 UM EXEMPLO DE APLICAÇÃO 39

4.2.1 CLASSE PONTO 39
4.2.2 CLASSE LINHA E CLASSE CIRCULO 40
4.2.3 INSTÂNCIAS DE CLASSE COMO VARIÁVEIS MEMBROS DE UMA CLASSE 41
4.2.4 A FUNÇÃO MEMBRO MOVER() 41
4.2.5 A FUNÇÃO MEMBRO DESENHAR() 42
4.2.6 CLASSE QUADRO 44
4.2.7 COMPATIBILIDADE DE TIPOS 46
4.2.8 CLASSE INCOMPLETAS - FUNÇÕES VIRTUAIS PURAS 46

4.3 O EXEMPLO GEOMÉTRICO MELHORADO 48

4.3.1 INICIALIZAÇÃO DE OBJETOS 50
4.3.2 FINALIZAÇÃO DE OBJETOS 52
4.4 HERANÇA MÚLTIPLA 53
4.4.1 AMBIGUIDADES EM HERANÇA MÚLTIPLA 61

Capítulo I - Introdução

Este capítulo contém uma introdução à história da linguagem C++, idéias que influenciaram o projeto da linguagem C++, bem como apresenta uma análise comparativa entre as linguagens C e C++.

1.1 Abstração de Dados

Abstração de Dados e Programação Orientada a Objetos representam um estilo de programação que oferece oportunidades para a melhoria na qualidade de software. Programação orientada a objetos, com abstração de dados como um fundamento necessário, difere enormemente dos outros estilos e metodologias de programação, uma vez que requer uma abordagem diferente para a resolução de problemas.

Programadores há muito reconhecem o valor de se organizar ítens de dados correlatos em construções de programas tais como *Record*s de Pascal e *struct*s de C, tratando-as posteriormente como unidades. Abstração de dados estende essa organização de forma a incorporar um conjunto de operações que possam ser executadas sob uma instância particular da estrutura. Usualmente, os elementos de dados e a implementação das operações que podem ser executadas sobre eles são mantidos privativos ou protegidos de forma a prevenir alterações indesejáveis. Ao invés de acessar os elementos de dados diretamente, os programas clientes (código do usuário) devem invocar as operações exportáveis de forma a produzir os resultados desejáveis.

Quando nós encapsulamos dados e operações que atuam sob esses dados desta maneira, estas estruturas encapsuladas comportam-se analogamente a tipos pré-construídos ou fundamentais como números inteiros ou de ponto flutuante. Pode-se, então, usá-las como caixas pretas que proporcionam uma transformação entre a entrada e a saída. Nós não necessitamos saber como o compilador trata os tipos fundamentais. Abstração de dados, em essência, nos permite criar novos tipos de dados, dai surgiu a idéia de chamá-los de Tipos Abstratos de Dados.

```
Encapsulamento + Proteção = Tipo Abstrato de Dados
```

1.2 Histórico

A Programação Orientada a Objetos não é nova; seus conceitos remontam há duas décadas atrás. A origem da programação orientada a objetos surge com as linguagens de programação Simula 67 e Smalltalk. Novo é o fluxo recente de interesse nesta importante e promissora metodologia. Foi dito que a programação orientada a objetos difere substancialmente dos estilos de programação com os quais nós estamos mais familiarizados, requerendo uma abordagem diferente para se resolver problemas, mas o que é programação orientada a objetos, e o que esta tem de diferente?

Quando se está programando orientado a objetos, um programador especifica o que fazer com um objeto antes de se concentrar no aspecto procedimental convencional de como algo deve ser feito. Programação orientada a objetos lida com a manipulação de objetos. Um objeto, por sua vez, pode representar quase tudo - um numero, uma string, um registro de paciente em um hospital, um trem ou uma construção gráfica como um retângulo ou alguma outra forma geométrica. Em essência, um objeto compreende os elementos de dados necessários para descrever os objetos, junto com o conjunto de operações permissíveis sobre esses dados. Pode-se ver que um objeto nada mais é do que uma instância particular de um tipo abstrato de dados, o qual foi projetado de acordo com um conjunto particular de regras.

Uma parte considerável do potencial da programação orientada a objeto resulta da utilização do mecanismo de herança - a idéia pela qual um programador começa com uma biblioteca de classes (tipos de objetos) já desenvolvida e a estende para uma nova aplicação adicionando novos elementos de dados ou operações (atributos) de forma a construir novas classes. Em outras palavras, ao invés de desenvolver uma nova aplicação escrevendo o código a partir do zero, um cliente herda dados e operações de alguma classe base útil, e então adiciona nova funcionalidade descrevendo como a nova classe difere da classe base. Ao adicionar novos elementos de dados ou funções, o programador não necessita modificar a classe base. Isto nos conduz a um dos benefícios da programação orientada a objetos: a reutilização de código.

O mecanismo de herança abre a possibilidade de ocorrência de *late* ou *dynamic binding* (amarração tardia ou amarração dinâmica), onde um programa cliente pode aplicar uma função a esse objeto sem saber a classe específica do objeto. Em tempo de execução, o sistema *run-time* determinará a classe específica do objeto e invocará a implementação correspondente da função.

Mas o que tudo isto significa, afinal de contas? Para começar a compreender estes novos conceitos, considere um programa que manipula estruturas de dados representando figuras geométricas. Talvez pudéssemos começar com uma classe Forma para representar as formas gerais. Através da herança, nos poderíamos projetar novas classes de Forma a representar formas mais especificas, como Círculos e Quadrados.

Imagine agora uma situação na qual nós necessitamos de um programa que ao invés de operar sob estruturas de dados especificas como círculos, possa lidar com

Formas genéricas cujos tipos particulares nós não conhecemos quando escrevemos o programa. Por exemplo, nós podemos querer que o programa desenhe um conjunto de Formas, tais com Círculos, Triângulos e Retângulos. Quando o programa estiver executando, o sistema run-time determinará a classe particular de cada objeto do tipo Forma e, utilizando late binding, chamará a função desenhar() apropriada para este.

1.3 A Linguagem C++

A linguagem C++ foi projetada e implementada por Bjarne Stroustrup, do AT&T Bell Laboratories, para se tornar a linguagem sucessora de C. C++ estende C através do uso de várias idéias emprestadas das linguagens de programação Simula 67 e Algol 68. Mesmo assim, C++ mantém total compatibilidade com C, isto é, qualquer programa feito em C também é um programa em C++. A figura 1 ilustra a árvore genealógica das linguagens de programação mais conhecidas.

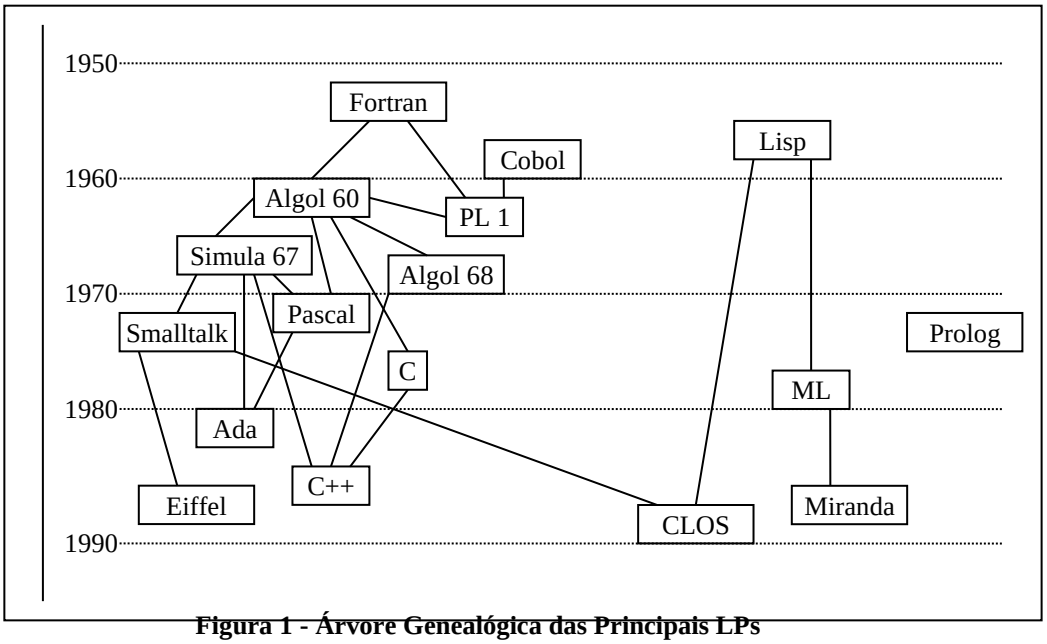


Figura 1 - Árvore Genealógica das Principais LPs

C++ também adiciona muitas novas facilidades a C, fazendo com que esta seja conveniente para um amplo leque de aplicações: de drivers de dispositivos às aplicações comerciais.

Muitos programadores concordam que C é uma linguagem compacta e eficiente. Contudo, ela possui deficiências, algumas das quais foram corrigidas na linguagem C++.

Por exemplo, C possui um único mecanismo de passagem de parâmetros, conhecido como passagem por valor. Quando se necessita passar um argumento para uma função por referência, deve-se utilizar um mecanismo obscuro e indutor de erros, que é a passagem de um ponteiro como parâmetro. C++ resolve este problema de uma forma elegante e eficiente.

C++ não apenas corrige muitas das deficiências de C, mas também introduz muitas características completamente novas que foram projetadas para suportar abstração de dados e programação orientada a objetos. Exemplos de tais características são:

• *Classes*, a construção básica que permite ao programador definir novos tipos de dados;

• *Variáveis membros*, que descrevem a representação de uma classe, e *funções membros*, que definem as operações permissíveis sob uma classe;

• *Overloading* (Sobrecarga) *de operadores*, que permite a definição de significados adicionais para muitos operadores, de modo que estes possam ser usados com os tipos de dados criados pelo programador, e *overloading de funções*, similar ao overloading de operadores, que permite reduzir a necessidade de criação de nomes para funções, tornando o código mais fácil de ler;

• *Herança*, que permite que uma classe derivada herde as variáveis membros e funções membros de sua classe base;

• *Funções virtuais*, que permitem que uma função membro herdada de uma classe base seja redefinida em uma classe derivada.

Capítulo II - Abstração de Dados

Este capítulo descreve as facilidades que C++ fornece para a declaração de novos tipos de dados, bem como as maneiras pelas quais uma estrutura de dados pode ser protegida, inicializada, acessada e, finalmente, destruída, através de um conjunto especifico de funções.

2.1 Introdução

As linguagens de programação tratam as variáveis de programa e constantes como instâncias de um tipo de dados. Um tipo de dado proporciona uma descrição de suas instâncias fornecendo ao compilador informações, tais como: quanto de memória alocar para uma instância, como interpretar os dados, e quais operações são permissíveis sob aqueles dados. Por exemplo, quando nós escrevemos uma declaração tal como *float x;* em C ou C++, estamos declarando uma instância chamada *x* do tipo de dado *float*. O tipo de dado *float* diz ao compilador para reservar, por exemplo, 32 bits de memória, para usar instruções de ponto flutuante para manipular esta memória, que operações tais como "somar" e "multiplicar" são aceitas e que operações como "módulo" e "shift" não são. Nós não temos que escrever esta descrição do tipo *float* - o implementador do compilador fez isto para nós. Tipos de dados como *float*, *int* e *char* são conhecidos com tipos de dados primitivos.

Algumas linguagens de programação têm características que nos permitem estender efetivamente a linguagem através da adição de novos tipos de dados.

Pode-se implementar novos tipos de dados na linguagem C++ através da declaração de uma classe. Uma classe é um novo modelo de dados definido pelo usuário que proporciona encapsulamento, proteção e reutilização. Uma classe tipicamente contém:

- Uma especificação de herança;
- Uma definição para a representação (adicional) de dados de suas instâncias (objetos);
- Definições para as operações (adicionais) aplicáveis aos objetos;

A melhor maneira de aprender sobre abstrações de dados em C++ é escrever um programa, e é isto que será feito nas próximas seções. Vamos começar em um território familiar, escrevendo um programa simples em C.

```
#include <stdio.h>

main ()
{
int a = 193;
int b = 456;
int c;
c = a + b + 47;
printf("%d\n",c);
}
```

Exemplo 1

Este programa declara três variáveis inteiras *a*, *b* e *c*, inicializando *a* e *b* com os valores 193 e 456, respectivamente. À variável inteira *c* é atribuído o resultado da adição *a*, *b* e o literal *47*. Finalmente, a função *printf()* da biblioteca padrão de C é chamada para imprimir o valor de *c*.

2.2.1 Problema Exemplo

Agora suponha que se deseja executar um cálculo similar, mas desta vez, a e b são números muito grandes, como a dívida interna do Brasil expressa em reais. Tais números são muito grandes para serem armazenados como *ints*, de modo que se nós tentássemos escrever *a = 25123654789456;* o compilador C emitiria uma mensagem de erro e o programa não seria compilado com sucesso. Inteiros grandes têm muitas aplicações práticas como criptografia, álgebra simbólica, e teoria dos números, onde pode ser necessário executar operações aritméticas com números de centenas ou mesmo milhares de dígitos.

2.2 Um Exemplo de Abstração de Dados em C++

Utilizando-se C++ é possível construir uma solução para o problema anterior de uma forma conveniente e elegante. Note como o programa do exemplo 1 é similar ao programa do Exemplo 2.

Muitas linguagens atualmente em uso, tais como C, COBOL, Fortran ,Pascal, e Modula-2 tornam difícil a abstração de dados. Isto porque a abstração de dados requer características especiais não disponíveis nestes linguagens. Afim de obter sentimento a respeito destas idéias, nós iremos analisar o programa do Exemplo 2.

As três primeiras instruções no corpo da função main() declaram três variáveis do tipo *BigInt*: *a*, *b* e *c*. O Compilador C++ necessita saber como criá-las, quanto de memória é necessário para alocar para elas e como inicializá-las.

A primeira e a segunda instruções são similares. Elas inicializam as variáveis *BigInt a* e *b* com constantes literais representando inteiros grandes escritos como strings de caracteres contendo somente dígitos. Sendo assim, o compilador C++ deve ser capaz de converter strings de caracteres em *BigInt*s.

```
# include "BigInt.h"

 main()
{
BigInt a = "25123654789456";
BigInt b = "456023398798362";
BigInt c;
c = a + b + 47;
c.print();
printf("\n");
}
```

Exemplo 2

A quarta instrução é mais complexa. Ela soma *a*, *b* e a constante inteira *47*, e armazena o resultado em *c*. O compilador C++ necessita ser capaz de criar uma variável *BigInt* temporária para reter a soma de *a* e *b*. E, então, deve converter a constante inteira *47* em um *BigInt* e somá-la com a variável temporária. Finalmente, deve atribuir o valor desta variável temporária a variável *c*.

A quinta instrução imprime *c* na saída padrão, através da função membro *print*. A última instrução chama a função *printf* da biblioteca padrão de C para imprimir o caracter de nova linha.

Apesar do corpo de main() não possuir nenhuma instrução adicional, o trabalho ainda não está finalizado. O compilador deve gerar código para que as variáveis *a*, *b* e *c*, e qualquer *BigInt* temporário sejam destruídos antes de deixar uma função. Isto assegura que a memória será liberada.

De uma forma resumida, o compilador necessita saber como:
• Criar novas instâncias das variáveis *BigInt*;
• Converter strings de caracteres e inteiros para *BigInt*s;
• Inicializar o valor de um *BigInt* com o de outro *BigInt*;
• Atribuir o valor de um *BigInt* a outro;
• Somar dois *BigInt*s;
• Imprimir *BigInt*s;
• Destruir *BigInt*s quando eles não são mais necessários.

2.2.1 Especificação e Implementação

Onde o compilador C++ obtém este conhecimento? Do arquivo *BigInt.h*, que é incluído na primeira linha do programa exemplo. Este arquivo contém a especificação de nosso novo tipo de dados *BigInt*. A especificação contém as informações que os programas que usam um tipo abstrato de dados, chamado de programas clientes (ou simplesmente, clientes) necessitam, afim de compilarem com êxito. Muitos dos detalhes de como um novo tipo funciona, conhecidos como a implementação, são mantidos em um arquivo à parte. Em nosso exemplo, este arquivo é chamado

BigInt.cpp. A Figura 2 mostra como a especificação e a implementação de um tipo abstrato de dados são combinados com o código fonte do cliente para produzir um programa executável.

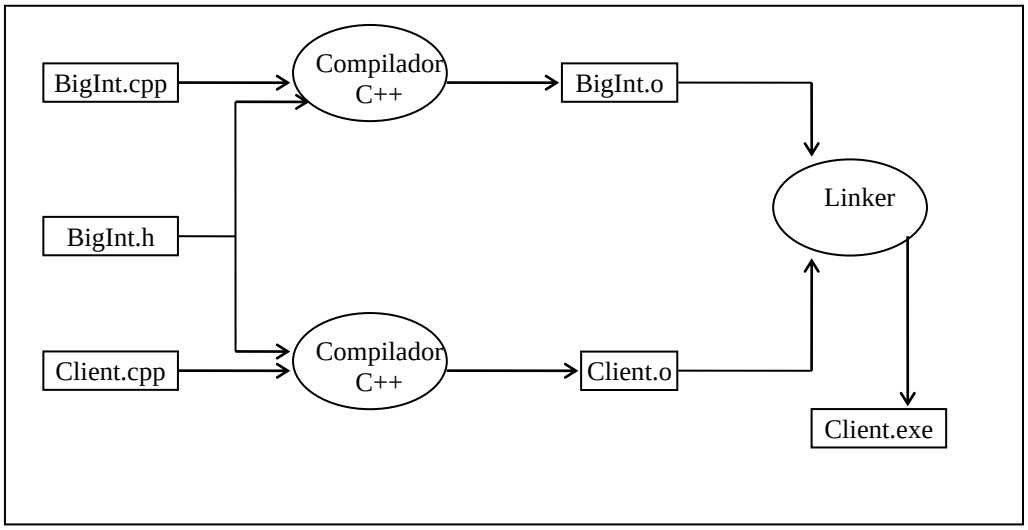


Figura 2 - Combinando Especificação e Implementação em um Programa Cliente

O objetivo de se separar o código de um tipo abstrato em uma especificação e uma implementação é ocultar os detalhes de implementação do cliente. Nós podemos, então, mudar a implementação e ter garantias que os programas clientes continuarão a trabalhar corretamente. Um tipo abstrato de dados bem projetado também oculta sua complexidade em sua implementação, tornando tão fácil quanto possível para os clientes utilizá-lo.

2.3 Um Exemplo de Especificação

2.3.1 Especificação da Classe BigInt

A instrução *#include "BigInt.h"* inclui a especificação da classe *BigInt* dentro do programa exemplo, de modo que se possa usar *BigInt*s no programa. Aqui está o conteúdo de *BigInt.h* (note que em C++, // começa um comentário que se estende até o fim da linha):

```
#include <stdio.h>
// este arquivo contem a especificação da classe BigInt

class BigInt {
// ...
        char* digitos;          // ponteiro para um array de digitos na memória
        unsigned ndigitos;      // numero de digitos
// ...
public:
        BigInt (const char*);                   // funcao construtora
        BigInt (unsigned n = 0);                // funcao construtora
        BigInt (const BigInt&);                 // funcao construtora utilizando copia
        void operator= (const BigInt&);         // atribuição
        BigInt operator+ (const BigInt&) const;     // operador adicao
        void print (FILE* f = stdout) const;        // função de impressao
        ~BigInt() { delete digitos; }               // destrutor

};
```

Programadores C entenderam muito pouco deste código.

Este é um exemplo de uma das mais importantes características de C++, a declaração de uma classe. Esta é uma extensão de uma construção que programadores em C já deveriam estar familiarizados: a declaração de uma *struct*.

A declaração de uma estrutura agrupa um certo número de variáveis, que podem ser de tipos diferentes, em uma unidade. Por exemplo, em C (ou C++) pode-se escrever:

```
struct BigInt {
        char* digitos;
        unsigned ndigitos;
};
```

Pode-se então declarar uma instância desta estrutura escrevendo-se:

```
struct BigInt a;
```

As variáveis membros da *struct*, *digitos* e *ndigitos*, podem ser acessados utilizando-se o operador ponto(.); por exemplo, *a.digitos*, acessa a variável membro *digitos* da *struct a*.

Relembre que em C pode-se declarar um ponteiro para uma instância de uma estrutura:

```
struct BigInt* p;
```

Neste caso, pode-se acessar as variáveis individuais utilizando-se o operador ->, por exemplo, *p->digitos*.

Classes em C++ trabalham de forma similar, e os operadores . e -> podem ser usados da mesma maneira para acessar as variáveis membro de uma classe. Neste exemplo, a classe *BigInt* tem duas variáveis membros *digitos* e *ndigitos*. A variável *digitos* aponta para um array de bytes, alocados na área de memória dinâmica, e contém os dígitos de um inteiro grande (um dígito por byte do array). Os dígitos são ordenados começando do dígito menos significativo no primeiro byte do array, e são números binários, não caracteres ASCII. A variável membro *ndigitos* contém o numero de dígitos de um inteiro. A figura 3 mostra o diagrama de uma instância desta estrutura para o número 654321.

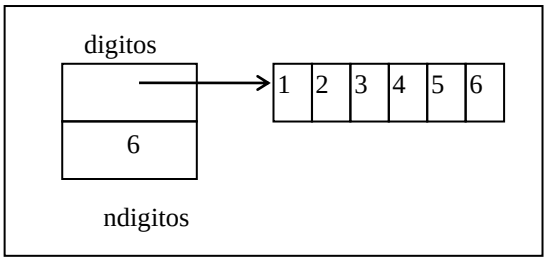


Figura 3 - Uma Instância da Classe *BigInt* Representando o Número "654321"

2.3.2 Ocultamento de Informação

Um programa cliente pode declarar uma instância da classe *BigInt* da seguinte forma:

BigInt a;

Temos agora um problema potencial: o programa cliente pode tentar, por exemplo, usar o fato de que *a.ndigitos* contém o número de dígitos no número *a*. Isto tornaria o programa cliente dependente de implementação da classe *BigInt*. Necessita-se, portanto, de uma forma de prevenção contra o acesso não autorizado a variáveis membros (representação) de uma classe. C++ permite que o usuário controle o acesso aos atributos de uma classe. A palavra reservada ***public:*** dentro de uma declaração de classe indica que os membros que seguem esta palavra reservada podem ser acessados por todas as funções. Já os membros que seguem a palavra reservada ***private:***, como *digitos* e *ndigitos* em nosso exemplo, só podem ser acessados pelas funções membros declaradas dentro da mesma classe. Há um terceiro tipo de controle determinado pela palavra chave ***protected:***, o qual será analisado posteriormente.

Restringir o acesso aos membros de uma classe é uma forma de aplicação do princípio de ocultamento de informação, este princípio pode ser estabelecido como:

> *Todas as informações a respeito de uma classe (módulo) devem ser privativas desta classe, a menos que uma informação específica seja declarada pública.*

Este princípio visa garantir que mudanças na implementação (representação) de uma classe não afetem os clientes desta classe. Por conseguinte, torna-se mais fácil

fazer modificações, porque o código que manipula as variáveis membros protegidas é localizado, e isto auxilia no processo de depuração.

2.3.3 Funções Membros

Como um programa cliente interage com as variáveis membros privadas de uma classe? Enquanto a construção *struct* de C permite apenas que variáveis sejam agrupadas juntas, a declaração *class* de C++ permite o agrupamento de variáveis e funções. Tais funções são chamadas de funções membros, e as variáveis privadas das instâncias de uma classe somente podem ser acessadas através das funções membros[1].

Sendo assim, um programa cliente só pode ler e modificar os valores das variáveis membros privativas de uma classe indiretamente, chamando as funções membros públicas de uma classe, como mostrado na Figura 4.

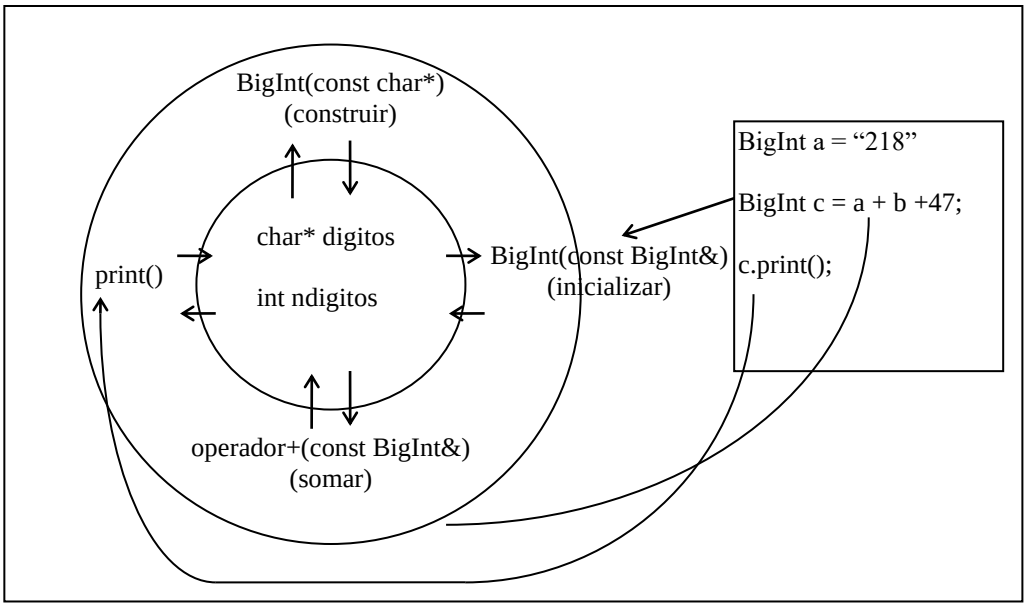


Figura 4 - Acesso a Variáveis Privadas através de Funções Membro Públicas

Por exemplo, a classe *BigInt* tem duas variáveis membros privativas, *digitos* e *ndigitos*, e sete funções membros públicas. A declaração destas funções parecerá estranha aos programadores C pelas seguintes razões:

• três funções têm o mesmo nome, *BigInt*;

• os nomes de funções *operator=*, *operator+* e *~BigInt* contém caracteres normalmente não permitidos em nomes de funções.

2.3.4 Overloading

[1] Estritamente falando, funções friend também podem acessar os membros privativos de uma classe, como será visto posteriormente.

Um identificador ou operador é sobrecarregado (overloaded) se simultaneamente denota duas ou mais funções distintas. Em geral, sobrecarga (overloading) é aceitável somente quando cada chamada de função é não ambígua, isto é, quando cada função pode ser identificada usando-se somente as informações de tipos disponíveis dos parâmetros. Por exemplo, em C++ pode-se declarar duas funções com o mesmo nome *abs*:

```
int abs(int);
float abs(float)
```

Pode-se então escrever:

```
x = abs (2);
y = abs (3.14);
```

A primeira instrução chamara *abs(int)*, e a segunda *abs(float)* - o compilador C++ sabe qual função *abs* deve usar porque *2* é um *int* e *3.14* é um *float*. Overloading reduz o numero de indentificadores de funções que um programador deve memorizar, e também elimina nomes artificiais para as funções.

Deve-se notar que, ao contrário de outras linguagens, como por exemplo Ada, C++ considera somente o número e o tipo dos argumentos de forma a determinar qual função será utilizada. Sendo assim, não se pode sobrecarregar *float sqr(float)* e *double sqr(float)*, por exemplo.

2.3.5 Funções com Argumentos Default

A especificação da classe *BigInt* contém as seguintes declarações de função:

```
BigInt (unsigned n = 0)      // funcao construtora
void print (FILE* f = stdout);     // função de impressao
```

Estas declarações demandam explicações adicionais. Além da declaração do tipo dos argumentos, estas instruções declaram argumentos default. Se você chamar uma destas funções sem argumento, C++ utiliza a expressão do lado direito do sinal = como default. Por exemplo, a chamada de função *print()* é idêntica a chamada *print(stdout)*. Ao se especificar defaults convenientes para as funções pode-se abreviar as chamadas de funções e torná-las mais legíveis. Também se reduz o número de funções que se necessita escrever.

Os argumentos defaults de uma função deverão ser os argumentos mais a direita em uma determinada função. Assim, a declaração

```
unsigned Maximo (unsigned a, unsigned b, unsigned c=0, unsigned d=0);
```

é legal, mas as declarações

```
unsigned Maximo (unsigned a, unsigned b=0, unsigned c);
unsigned Maximo (unsigned a=0, unsigned b, unsigned c);
```

não são.

2.3.6 Chamando Funções Membros

Considere agora a penúltima linha do programa do exemplo 2:

c.print();

Funções membros são chamadas de uma maneira análoga a forma como variáveis membros são acessadas em C, isto é, utilizando-se o operador . ou o operador ->. Uma vez que c é uma instância da classe *BigInt*, a notação *c.print()* chama a função membro *print* da classe *BigInt* para imprimir o valor atual de *c*. Se nós declarássemos um ponteiro para um *BigInt*

```
BigInt* p;
p = &c;
```

então a notação *p->print()* chamaria a mesma função. Esta notação assegura que uma função membro pode operar somente sobre instâncias da classe para a qual esta foi definida. Em C++, várias classes diferentes podem ter funções membros com o mesmo nome.

2.3.7 Construtores

Uma das coisas que o compilador C++ necessita saber sobre o tipo abstrato BigInt é como criar novas instâncias da classe *BigInt*. Pode-se informar ao compilador C++ como isto deve ser feito definindo-se uma ou mais funções membros especiais chamadas de construtores. Uma função construtora possui o mesmo nome de sua classe. Quando o programa cliente contém uma declaração como,

BigInt a = "123";

O compilador C++ reserva espaço para as variáveis membros *digitos* e *ndigitos* para uma instância da classe *BigInt* e então chama a função construtora *a.BigInt("123")*. É responsabilidade do implementador da classe *BigInt* escrever a função *BigInt()* de modo que ela inicialize as instâncias corretamente. Neste exemplo, *BigInt("123")* aloca três bytes para memória dinâmica, ajusta *a.digitos* para apontar para esta memória, modifica os três bytes para [3,2,1], e atribui o valor três para *a.ndigitos*. Isto criará uma instância da classe *BigInt* que é inicializada com o valor 123.

Se uma classe tem uma ou mais funções construtoras (funções construtoras podem ser sobrecarregadas), C++ garante que uma função será chamada para inicializar cada instância da classe. Um usuário de uma classe, tal como *BigInt*, não tem de se lembrar de chamar uma função de inicialização separadamente para cada *BigInt* declarado.

Uma classe pode ter um construtor que não requer nenhum argumento, ou pode ter uma função construtora com **defaults** para todos os seus argumentos. Nós chamamos esta função de construtor **default**. Este é o construtor que C++ chama para inicializar a variável *b* mostrada abaixo:

BigInt b;

C++ também utiliza o construtor **default** para inicializar os elementos individuais de um vetor de instâncias de classe, por exemplo:

BigInt a[10];

C++ chamará o construtor **default** da classe *BigInt* 10 vezes, uma vez para a[0], a[1], ..a[9].

Pode-se também escrever a declaração:

BigInt a = "123";

usando uma lista de argumentos, como a seguir:

BigInt a ("123"); // mesmo que BigInt a = "123";

De fato, deve-se utilizar uma lista de argumentos para declarar uma instância de uma classe que demande mais de um argumento. Por exemplo, se nós tivéssemos um classe BigComplex que tivesse o seguinte construtor:

BigComplex (BigInt re, BigInt im);

nós poderíamos declarar uma instância da classe *BigComplex* chamada de *x* e inicializá-la para (12,34) através de:

BigComplex x(12,34);

2.3.8 Construtores e Conversão de Tipos

Para compilar o exemplo 2, o compilador C++ necessita saber como converter uma string de caracteres como *"25123654789456"*, ou um inteiro como *47*, para um *BigInt*. Construtores também são usados com esse propósito. Quando o compilador C++ encontra uma declaração como:

BigInt c = a + b + 47;

ele reconhece que um *int* (*47*) deve ser convertido para um *BigInt* antes que a adição se realize. Sendo assim, ele necessita verificar se um construtor *BigInt (unsigned)* está declarado. Se este existir, uma instância temporária da classe *BigInt* será criada através de uma chamada a este construtor com o argumento *47*. Se o construtor apropriado não está declarado, o compilador sinalizará um erro. A classe *BigInt* possui dois construtores *BigInt(char*)*, *BigInt (unsigned)*, de modo que se pode

utilizar strings de caracteres ou inteiros, e o compilador C++ automaticamente chamará o construtor apropriado para realizar a conversão de tipos.

Pode-se também chamar o construtor explicitamente para forçar a conversão de tipo. Por exemplo, C++ processaria as declarações:

unsigned long i = 4000000000;
BigInt c = i + 1000000000;

adicionando-se *i* a constante *1000000000*, e então convertendo o resultado para um *BigInt* de forma a inicializar *c*. Infelizmente, a soma de *i* e *1000000000* provavelmente causará um erro de **overflow**. Pode-se, contudo, evitar este problema convertendo-se explicitamente *i* ou *1000000000* para um *BigInt*:

BigInt c = BigInt (i) + 1000000000;

ou

BigInt c = BigInt (1000000000) + i;

Pode-se também escrever isto utilizando o **cast** de C e C++:

BigInt c = (BigInt) i + 1000000000;

ou

BigInt c = i + (BigInt) 1000000000;

A notação de chamada do construtor é mais indicativa do que está ocorrendo do que a notação **cast**, e é também mais geral desde que permite que se utilize construtores que requerem múltiplos argumentos.

2.3.9 Construtores e Inicialização

Compilar o exemplo 2 também requer que o compilador C++ saiba como inicializar um *BigInt* com o valor de outro *BigInt*, por exemplo:

BigInt c = a + b + 47;

A instância *BigInt c* deve ser inicializada com o valor da instância temporária da classe *BigInt* que contém o valor da expressão *a + b + 47*. C++ fornece um construtor pré-definido que copia objetos na inicialização. Deve-se, contudo, tomar cuidado ao utilizar este construtor pré-definido quando as instâncias da classe alocam dinamicamente algum dos seus elementos. Isto decorre do fato dele fazer cópia **bitwise** de um objeto sobre o outro, não copiando os elementos alocados dinamicamente. Tal fato pode gerar, portanto, sinonímias e efeitos colaterais.

Pode-se controlar como C++ inicializa instâncias da classe *BigInt* definido-se uma função construtora de copia *BigInt (const BigInt&)*[2]. No exemplo, este construtor aloca memória para uma nova instância da classe e faz uma cópia do conteúdo da instância passada como argumento.

[2] O argumento de BigInt& é um exemplo de uma referência em C++, que será posteriormente descrita.

Quando se programa em C++, é importante entender a distinção entre inicialização e atribuição. Inicialização ocorre em três contextos:

• declarações com inicializadores (como descrito acima);
• argumentos formais de funções;
• valores de retorno de funções;

Atribuições ocorrem em expressões (não em declarações) que utilizam o operador =. Por exemplo,

BigInt c = a + b + 47;

uma chamada ao construtor *BigInt (BigInt&)* inicializa a variável *c*. Contudo em:

BigInt c;
c = a + b + 47;

uma chamada ao construtor *BigInt (unsigned n=0)* inicializa a variável *c* para zero, e a segunda instrução atribui o valor da variável temporária *BigInt* a *c*.

2.3.10 Overloading de Operadores

C++ também deve ser capaz de adicionar dois *BigInt*s e atribuir o valor a um novo *BigInt* afim de compilar o Exemplo 2. Pode-se definir funções membros chamadas de **add** e **assign** para fazer isto, contudo, esta solução tornaria a escrita de expressões aritméticas deselegantes. C++ permite a definição de significados adicionais para muitos de seus operadores, incluindo + e =, sendo assim estes podem significar "**add**" e "**assign**" quando aplicados a *BigInt*s. Isto é conhecido como **overloading** de operadores, e é similar ao conceito de **overloading** de funções.

Realmente, muitos programadores já estão familiarizados com esta idéia porque os operadores de muitas linguagens de programação, incluindo C, já são sobrecarregados. Por exemplo, em C:

int a,b,c;
float x,y,z;
c = a + b;
z = x + z;

Os operadores + e = realizam coisa diferentes nas duas últimas instruções: a primeira faz adição e atribuição de inteiros e a segunda faz adição e atribuição de pontos flutuante. Overloading de operadores é simplesmente uma extensão disto.

C++ reconhece um nome de função tendo a forma *operator@* como um overloading do símbolo @. Pode-se sobrecarregar os operadores + e = declarando-se as funções membros *operator+* e *operator=*, como foi feito na classe *BigInt*:

void operator= (const BigInt&); *// atribuição*

BigInt operator+(const BigInt&) const; // operador adição

Pode-se chamar estas funções utilizando-se a notação usual para chamada de funções membros utilizando-se apenas o operador:

BigInt a,b,c;
c = operator+(b);
c = a + b

As duas últimas linhas são equivalentes.

Deve-se ressaltar que a sobrecarga dos operadores não muda os significados pré-definidos, apenas dá um significado adicional quando utilizado com instâncias da classe. A expressão *2 + 2* continua dando *4*. A precedência dos operadores também não é alterada.

Pode-se sobrecarregar qualquer operador de C++, exceto o operador de seleção (.) e o operador de expressão condicional (*?:*).

A utilização da sobrecarga de operadores é recomendada apenas quando o operador sugere fortemente a função que este executa. Um exemplo de como sobrecarga NÃO deve ser usada é sobrecarregar o operador + para executar uma subtração.

2.3.11 Destrutores

O compilador C++ necessita saber, ainda, como destruir as instâncias da classe *BigInt*. Pode-se informar ao compilador como fazer isto definido-se um outro tipo especial de função chamada destrutor. Uma função destrutora tem o mesmo nome de sua classe prefixado pelo caracter ~. Para a classe *BigInt*, esta é a função membro *~BigInt().*

Deve-se escrever a função destrutora de uma forma que esta finalize apropriadamente instâncias da classe. No exemplo apresentado, isto significa liberar a memória dinâmica alocada para a instância da classe pelo construtor.

Se uma classe tem uma função destrutora, C++ garante que ela será chamada para finalizar toda instância da classe quando esta não for mais necessária. Isto significa dizer que o usuário não necessita chamar explicitamente o destrutor da classe, o sistema se encarrega de fazer isto, eliminando assim uma fonte possível de erros de programação.

Diferente dos construtores, os destrutores não têm nenhum argumento, e assim não podem ser sobrecarregados. Por conseguinte, uma classe só possui um único destrutor.

Capítulo III - Exemplo de Implementação

3.1 Introdução

O capítulo anterior descreve um programa cliente em C++ que utiliza a classe *BigInt* e a especificação desta classe. O presente capítulo irá descrever a implementação da classe *BigInt*, onde todo o trabalho real é feito. Ao se descrever a implementação da classe, várias características novas de C++ serão encontradas e explicadas: o operador de resolução de escopo, tipos constantes, funções membros constantes, referências, os operadores *new* e *delete*, funções *friend* e funções *inline*.

Como foi dito anteriormente a implementação de uma classe contém os detalhes de *como* ela trabalha.

3.2 Implementação de uma Classe BigInt

A implementação requer as informações contidas na especificação, sendo assim a primeira linha no arquivo *BigInt.cpp* é:

#include "BigInt.h";

Uma vez que a implementação e os programas clientes são compilados com a mesma especificação, o compilador C++ assegura uma interface consistente entre elas.

3.2.1 O construtor BigInt(const char)*

A classe *BigInt* tem três construtores, um para declarar instâncias de um *BigInt* a partir de string de caracteres, um para criar instâncias de um inteiro não negativo (um *unsigned*) e um para inicializar um *BigInt* com outro. Aqui está a implementação do primeiro construtor:

```
BigInt:: BigInt (const char* digitString)
{
        unsigned n = strlen (digitString);
        if (n!=0) {
                digitos = new char [ndigitos = n ];
                char * p = digitos;
                const char* q = &digitString[n];
                while (n--) *p++ = *--q - '0';    // converte o digito ASCII para binario
        }
        else {    // string vazia
                digitos = new char[ndigitos = 1];
                digitos[0] = 0;
        }
}
```

Se a string é vazia esta é tratada como um caso especial e cria-se um *BigInt* inicializado para zero.

3.2.2 O Operador Resolução de Escopo

A notação *BigInt::BigInt* identifica *BigInt* como uma função membro da classe *BigInt*, isto se deve ao fato de que várias classes podem ter funções membros com o mesmo nome. O operador *::* é conhecido como operador de resolução de escopo, e pode ser aplicado a funções membros e variáveis membros. Logo, *BigInt::print()* refere-se a função membro *print()* que é membro da classe *BigInt*, e *BigInt::digitos* refere-se a variável membro *digitos* que é membro da classe *BigInt*.

3.2.3 Constantes

Um programador C estará familiarizado com o uso do tipo *char** para argumentos que são strings de caracteres, mas o que é uma *const char**? Em C++, pode-se usar a palavra reservada *const* para indicar que uma variável é uma constante, e portanto não pode ser modificada por um comando de atribuição (=). Quando utilizado em uma lista de argumentos como acima, isto previne que o argumento possa ser modificado pela função. Sendo assim, se tentarmos adicionar a seguinte instrução:

```
digitString[0] = 'x';
```

ao construtor, o compilador C++ emitirá uma mensagem de erro. Isto evita erros de programação muito comuns.

Recomenda-se a utilização de *const* para argumentos ponteiros (referências) que não devem ser modificados pelas funções.

Deve-se também empregar *const* ao invés de se utilizar a diretiva de pré processamento *#define* quando se deseja definir constantes simbólicas:

```
const double PI = 3.1415926;
```

A utilização de const permite ao compilador verificar o tipo do valor constante, além de permitir colocar a definição do identificador constante em uma classe, bloco ou função.

Deve-se notar que *const* restringe a maneira pela qual um objeto pode ser usado.

Quando se utiliza um ponteiro, dois objetos estão envolvidos; o próprio ponteiro e o objeto apontado. Quando se deseja que o objeto apontado, mas não o ponteiro, fique inalterado, deve-se prefixar a declaração do ponteiro com a palavra reservada *const*. Por exemplo:

```
const char* pc = "Nome";     // ponteiro para uma constante
pc[3] = 'a';                 // erro
pc = "teste"                 // OK
```

Para declarar um ponteiro constante, ao invés do objeto apontado, o operador **const* deve ser utilizado. Por exemplo:

```
char *const cp = "Nome";      // ponteiro constante
cp[3] = 'a';                  // OK
cp = "Teste";                 // erro
```

Para fazer ambos objetos constantes eles devem ser declarados *const*. Por exemplo:

```
const char *const cp = "Nome";    // ponteiro constante para objeto constante
cp [3] = 'a';                     // erro
cp = "Teste";                     // erro
```

O endereço de uma constante não pode ser atribuído a um ponteiro irrestrito (um ponteiro que não aponta para um objeto constante) porque isto permitiria que o valor do objeto fosse mudado. Por exemplo:

```
int a = 1;
const int c = 2;
const int* p1 = &c;     // OK
const int *p2 = &a;     // OK
int* p3 = &c;           // erro, pois
*p3 = 7;                // mudaria o valor de c
```

3.2.4 Funções Membros Constantes

Na declaração da classe *BigInt*, mostrada na seção 2.3, pode-se notar que a palavra reservada *const* aparece depois da lista de argumentos na declaração das funções membros *operator+()* e *print()*.

```
BigInt operator+(const BigInt&) const;      // operador adicao
void print (FILE* f = stdout) const;        // funcao de impressao
```

O que isto significa? Na seção 2.3.6 foi explicado que funções membros são aplicáveis a instâncias (objetos) de sua classe utilizando os operadores . e ->. Por exemplo:

```
BigInt c = "29979250000";   // velocidade da luz (cm/s)
c.print();
BigInt* cp = &c;
cp-> print();
```

A palavra reservada *const* depois da lista de argumentos de uma função membro informa ao compilador que a função membro é uma função membro constante: uma função membro que não modifica o objeto sobre a qual é aplicada, de modo que esta função pode ser aplicada a objetos constantes:

```
const BigInt c = "29979250000";   // velocidade da luz (cm/s)
c.print();
```

```
const BigInt* cp = &c;
cp-> print();
```

O compilador C++ diria que estas chamadas a *print()* são ilegais caso se omitisse a palavra reservada *const* depois da lista de argumentos na declaração e definição de *print()*.

Além disso, o compilador C++ também verifica se uma função membro constante não modifica nenhuma das variáveis membros das instâncias da classe. Sendo assim, por exemplo, se a definição da função membro *BigInt::print()* contivesse a instrução:

```
ndigitos--;
```

o compilador indicaria um erro, desde que isto muda o valor de uma das variáveis membros da classe *BigInt*.

3.2.5 O Operador New

Utiliza-se o operador *new* para alocução de memória dinâmica, necessária, por exemplo para manter os dígitos de um *BigInt*. Em C, para fazer isto, utilizaria-se a função padrão da biblioteca C, chamada *malloc()*. O operador *new* tem duas vantagens:

• Ele retorna um ponteiro para o tipo de dado apropriado. Enquanto que, para se alocar espaço para as variáveis membros de uma *struct BigInt* em C, teríamos que escrever:

```
struct BigInt* p;
p = (struct BigInt*) malloc(sizeof (struct BigInt));
```

em C++, nós podemos escrever simplesmente:

```
BigInt* p;
p = new BigInt;
```

• A segunda vantagem é que ao se utilizar o operador *new* para alocar uma instância de uma classe, o construtor desta classe é automaticamente chamado de modo a inicializar o objeto.

3.2.6 Declaração em Blocos

Programadores em C podem ter observado que a declaração de *p* parece estar mal colocada em

```
if (n!=0) {
        digitos = new char[ndigitos = n];    // uma instrução
```

```
char* p = digitos;                    // uma declaração !!!!!!
```

uma vez que esta aparece depois da primeira instrução do bloco. Em C++, uma declaração pode aparecer em qualquer lugar dentro de um bloco desde que a variável seja declarada antes de ser utilizada. A declaração é efetiva até o fim do bloco em que ela ocorre. Pode-se frequentemente melhorar a legibilidade de um programa colocando-se as declarações de variáveis próximas aos lugares onde são empregadas.

3.2.7 O Construtor BigInt (unsigned)

A implementação do construtor *BigInt(unsigned)*, o qual cria um *BigInt* de um inteiro positivo, é a seguinte:

```
BigInt:: BigInt(unsigned n)
{
        char d[3*sizeof(unsigned)+1];       // buffer para digitos decimais
        char *dp = d;                       // ponteiro para o proximo digito decimal
        ndigitos = 0;
        do {                            // converte inteiros para digitos decimais
                *dp++ = n%10;
                n /= 10;
                ndigitos++;
        } while (n > 0);
        digitos = new char[ndigitos];       // aloca espaco para digitos decimais

        for (register i=0; n<ndigitos, i++) digitos[i] = d[i];
}
```

Este construtor trabalha convertendo um argumento inteiro para dígitos decimais no **array** temporário d. Sabe-se, então, quanto espaço alocar para um *BigInt*, e é possível, portanto, alocar a memória dinâmica utilizando-se o operador *new*, e copiar os dígitos decimais do **array** temporário para esta área.

3.2.8 O Construtor Cópia

A função do construtor cópia é copiar o valor de um *BigInt* passado como argumento para uma nova instância de classe *BigInt*:

```
BigInt::BigInt(const BigInt& n)
{
        unsigned i = n.ndigitos;
        digitos = new char[ndigitos = i];
        char* p = digitos;
        char * q = n.digitos;
        while (i--) *p++ = *q++;
}
```

É importante observar o acesso a *n.ndigitos* dentro do construtor de cópia. Se *n* fosse um parâmetro de outra classe diferente de *BigInt*, o acesso a estrutura privada de *n* seria ilegal. Contudo, como o parâmetro é da própria classe *BigInt*, C++ permite que a sua estrutura interna seja acessada. Por conta disso, considera-se que C++ é mais orientado a classes do que a objetos.

Também cabe destacar que a função construtora de cópia faz uso de uma referência, uma importante característica de C++ que será descrita a seguir.

3.2.9 Referências

O parâmetro formal da função membro *BigInt(const BigInt&)* é um exemplo de uma referência em C++. Referências em C++ objetivam superar a ausência de mecanismos de passagem de parâmetros por referência em C.

Afim de entender o que isto significa, suponha que se deseja escrever uma função chamada *inc()* que adiciona um ao seu argumento. Se esta função fosse escrita em C, seria da seguinte forma:

```
void inc (int x)
{
        x++;
}
```

e fosse utilizada pelo seguinte segmento de programa

```
int y = 1;
inc(y);
printf("%d\n",y);
```

o programa imprimiria um 1, e não 2. Isto porque em C o valor *y* é copiado dentro do parâmetro formal *x* e a instrução *x++* incrementa esta cópia, deixando o valor de *y* inalterado. Este mecanismo de passagem de parâmetros é conhecido com passagem por valor.

A função *inc()* deve ser escrita em C da seguinte forma:

```
void inc (int* x)
{
        *x++;
}
```

e o segmento de código que utiliza essa função deve ser o seguinte:

```
int y = 1;
inc(&y);
printf("%d\n",y);
```

Note que três modificações devem ser feitas:
• o tipo do argumento da função deve mudae de *int* para *int**;
• cada ocorrência do argumento no corpo da função deve mudar de *x* para **x*;
• cada chamada da função deve mudar de *inc(y)* para *inc(&y)*.

Utilizando-se a passagem por referência de C++, a função *inc()* poderia ser escrita da seguinte forma:

```
void inc(int& x)
{
        x++;
}
```

e o segmento de código que utiliza esta função seria:

```
int y = 1;
inc(y);
printf("%d\n",y);
```

A utilização do mecanismo de passagem de parâmetros por referência simplifica, em muitos casos, a função e a utilização desta função.

Pode-se pensar em uma referência como sendo um ponteiro que é automaticamente derreferenciado quando utilizado. Pode-se declarar variáveis que são referências. Por exemplo:

```
int i;
int* p = &i;    // um ponteiro para i
int& r = i;     // uma referencia para i
r++;            // incrementa o valor de i
```

Você pode utilizar *r* em lugar de **p* e *&r* em lugar de *p*. As únicas diferenças são:

• Uma referência deve ser inicializada em sua declaração.
• Uma referencia não pode ser apontada para um objeto diferente daquele para o qual ela foi iniciada.

Sendo assim:

```
int j, i;
int* p = &i;    // um ponteiro para i
int& r = i;     // uma referencia para i
*p = 1;         // ajusta i para 1
j = *p;         // ajusta j para 1
r = 2;          // ajusta i para 2
j = r;          // ajusta j para 2
p = &j;         // ajusta p para apontar para j
```

&r = &j; // ilegal

Uma referência é um nome alternativo (apelido) para um objeto que proporciona uma sintaxe mais adequada e segura que a de ponteiros, contudo há situações que demandam o emprego de ponteiros.

3.2.10 O Operador Adição da Classe BigInt

A implementação da função membro *operator+()* é a seguinte:

```
BigInt BigInt::operator+(const BigInt& n) const
{
        unsigned maxDigitos = (ndigitos>n.ndigitos ? ndigitos : n.ndigitos) + 1;
        char* sumPtr = new char [ maxDigitos ];
        BigInt sum (sumPtr, maxDigitos);          // deve-se definir este construtor
        unsigned i = maxDigitos - 1;
        unsigned carry = 0;
        while (i--) {
            *sumPtr = /* proximo digito de this */ + /* proximo digito de n */ + carry;
             if (*sumPtr >= 10) {
                        carry = 1;
                        *sumPtr –=10;
             }
             else carry = 0;
             sumPtr ++;
        }
        if (carry)                        // verifica se numero de digitos cresceu
            *sumPtr = 1;
             return sum;
        else {                            // elimina casa adicional
           i = maxDigitos - 1;
            char* p = new char [i];
            char * q = sum.digitos;
            while (i--) *p++ = *q++;
           BigInt s (p, maxDigitos - 1);
           return s;
        }
}
```

A adição de dois *BigInt*s é feita utilizando o método aprendido nas escolas de 1° grau; adiciona-se os dígitos de cada operando, da direita para esquerda, começando do dígito mais a direita, e também somando-se um possível transbordo (**carry**) da coluna anterior.

Há dois problemas quando se escreve a função membro *operator+()*:

• Primeiro, há necessidade de se declarar uma instância da classe *BigInt* chamada *sum* que conterá o resultado da adição, o qual será deixado em um **array** apontado por *sumPtr*. Deve-se utilizar um construtor para criar uma instância de *BigInt* a partir de *sumPtr*, mas nenhum dos construtores disponíveis até agora é capaz de executar esta operação; sendo assim se faz necessário escrever um outro construtor para tratar esta nova situação.

Este novo construtor recebe como argumentos um **array** contendo os dígitos e os números de dígitos do **array** e cria um objeto da classe *BigInt* a partir desses argumentos. Deve-se ressaltar que não é conveniente que os programas clientes tenham acesso a essa função dependente de implementação, sendo assim nós a declaramos na parte privada da classe *BigInt*, de modo que ela só possa ser acessada pelas funções membros da classe *BigInt*. Logo, a declaração:

BigInt(char , unsigned);*

é acrescentada antes da palavra *public:* na declaração da classe *BigInt* no arquivo *BigInt.h*, e a implementação deste construtor é adicionada ao arquivo *BigInt.cpp*:

```
BigInt:: BigInt(char* d, unsigned n)
{
        digitos = d;
        ndigitos = n;
}
```

• O segundo problema é que a operação de percorrer os dígitos dos operandos na instrução

**sumPtr = /* proximo digito de this */ + /* proximo digito de n */ + carry;*

pode se tornar complicada porque um dos operandos pode conter um número menor de dígitos que o outro. Neste caso deve-se completá-lo com zeros mais a esquerda. Este mesmo problema aparecerá na implementação das operações de subtração, multiplicação e divisão, logo é relevante encontrar uma solução adequada para este problema. Uma boa solução é utilizar um tipo abstrato de dados !

Sendo assim, uma nova classe, chamada *SeqDigitos*, é definida de forma a manter uma trilha de qual dígito em um certo *BigInt* está sendo processado. A classe *SeqDigitos* tem um construtor que recebe como um argumento uma referência para um *BigInt* e inicializa uma *SeqDigitos* para esse *BigInt*. A classe *SeqDigitos* também tem uma função membro que retorna o próximo dígito do *BigInt* cada vez que ela é chamada, começando no dígito mais a direita (menos significativo). A declaração da classe *SeqDigitos* e a implementação de suas funções membros são as seguintes:

```
class SeqDigitos {
        char* dp;               // ponteiro para o digito corrente
        unsigned nd;            // numero de digitos restantes
public:
        SeqDigitos (const BigInt& n )     // construtor
        {
                dp = n.digitos;
                nd = n.ndigitos;
        }
        unsigned operator++( )          // retorna o digito atual e avança para o
                                        // próximo digito
        {
                if (nd == 0) return 0;
                else {
                        nd--;
                        return *dp++;
                }
        }
};
```

Pode-se agora declarar uma instância da classe *SeqDigitos* para cada um dos operandos e utilizar o operador ++ quando se necessita ler o próximo dígito.

Como esses dois problemas resolvidos, a implementação da função membro *operator+()* é a seguinte:

```
BigInt BigInt::operator+(const BigInt& n ) const
{
        unsigned maxdigitos = (ndigitos>n.ndigitos ? ndigitos : n.ndigitos) + 1;
        char* sumPtr = new char[maxdigitos];
        BigInt sum (sumPtr, maxdigitos);      // aloca memória para sum
        SeqDigitos a (*this);           // mais sobre a variável this posteriormente
        SeqDigitos b (n);
        unsigned i = maxdigitos - 1;
        unsigned carry = 0;
        while (i--) {
                *sumPtr = (a++) + (b++) + carry;
                if (*sumPtr >=10) {
                        carry = 1;
                        *sumPtr -=10;
                }
                else carry = 0;
                sumPtr++;
        }
        if (carry)                      // verifica se numero de digitos cresceu
            *sumPtr = 1;
             return sum;
        else {                          // elimina casa adicional
           i = maxDigitos - 1;
            char* p = new char [i];
            char * q = sum.digitos;
            while (i--) *p++ = *q++;
           BigInt s (p, maxDigitos - 1);
           return s;
        }
}
```

Deve-se observar que ao sobrecarregar o operador ++ e -- para uma classe, C++ não faz nenhuma distinção entre a forma pré-fixada e pós-fixada desses operadores.

3.2.11 Funções Friend

Você deve estar se perguntando como o construtor *SeqDigitos(const BigInt&)* é capaz de acessar os membros privativos *digitos* e *ndigitos* da classe *BigInt*. De fato, isso não é possível. Sendo assim há a necessidade de se conceder acesso a essas variáveis privativas à função construtoras da classe *SeqDigitos*, mas somente a essa função e nenhuma outra. C++ proporciona uma maneira de fazer isto - tornando o

construtor *friend* da classe *BigInt*, isto pode ser feito adicionando-se a seguinte declaração a especificação da classe *BigInt*:

friend SeqDigitos::SeqDigitos(const BigInt&);

Pode-se também fazer todas as funções membros de uma classe *friend* de outra declarando um classe inteira como um *friend*. Por exemplo, pode-se tornar todas as funções membros da classe *SeqDigitos friend*s da classe *BigInt* declarando-se na especificação da classe *BigInt*

friend class SeqDigitos;

3.2.12 A Palavra Reservada This

Retornando a implementação da função *BigInt::operator+()*, observa-se a utilização da variável ponteiro *this* na seguinte declaração:

*SeqDigitos a (*this);*

C++ automaticamente declara uma variável chamada de *this* em toda a função membro de uma classe e a inicializa para apontar para a instância da classe a qual a função membro foi aplicada. Sendo assim, na função membro *operator+()* da classe *BigInt*, *this* é implicitamente declarada como:

BigInt this;*

e quando *operator+()* é chamada, escrevendo-se uma expressão como *b + 47*, onde *b* é um *BigInt*, *this* é automaticamente inicializada para apontar para *b*. Logo, o efeito da declaração *SeqDigitos a(*this)* na função *operator+()* é criar uma instância de *SeqDigitos* para o operando da esquerda do *operator+()*, neste caso *b*.

Analogamente, quando uma função membro tal como *BigInt::print()* é chamada por uma expressão como *c.print()*, onde *c* é um *BigInt*, *this* é inicializado para apontar para *c*.

3.2.13 O Operador Atribuição da Classe BigInt

O objetivo do operador atribuição (=) é copiar o valor de um *BigInt* (o argumento da direita) para outro (o argumento da esquerda). Embora o operador de atribuição seja similar ao construtor cópia *BigInt(const BigInt&)*, há uma importante diferença: o construtor copia o valor para uma instância não inicializada de *BigInt*, enquanto o operador atribuição copia o valor dentro de uma instância inicializada de *BigInt*, isto é, uma que já contém um valor.

```
void BigInt::operator=(const BigInt& n)
{
    if (this == &n) return;      // para manipular x = x corretamente
```

```
        delete digitos;
        unsigned i = n.digitos;
        digitos = new char[n.digitos=i];
        char* p = digitos;
        char* q = n.digitos;
        while (i--) *p++ = *q++;
}
```

Deve-se ressaltar que o corpo da função *operator=()* é idêntico ao do construtor de cópia *BigInt (const BigInt&)*, exceto as duas primeiras instruções. A primeira instrução verifica se o endereço do operando da esquerda é o mesmo do operando da direita. Se for, nada necessita ser feito. Caso contrário, a memória dinâmica para os dígitos do operando da esquerda é liberada, chamando-se o operador *delete*.

3.2.14 A Função Membro BigInt::print()

A implementação da função membro constante *print()* é simples e direta:

```
void BigInt::print(FILE* f) const
{
        for (int i = ndigitos-1, i>=0, i--) fprintf(f, "%d", digitos[i]);
}
```

3.2.15 O Destrutor da Classe BigInt

A única coisa que a função destrutora *~BigInt()* da classe *BigInt* deve fazer é liberar a memória dinâmica alocada pelos construtores:

```
BigInt::~BigInt()
{
        delete digitos;
}
```

Isto é feito utilizando-se o operador *delete* de C++, o qual, neste caso, libera a memória dinâmica que é apontada por *digitos*. Ao se utilizar o operador *delete* para desalocar uma instância (objeto) de uma classe que tem uma função destrutora definida, o destrutor é automaticamente chamado para finalizar a instância da classe antes que a memória seja liberada. Por exemplo:

```
        BigInt* a = new BigInt("9834567843");
        //...
        delete a;
```

automaticamente executar-se-á *a->~BigInt()*, que desalocará a memória apontada por *digitos* antes de desalocar a memória para as variáveis membros das instâncias da classe *BigInt*.

A função destrutora deve ser chamada para cada elemento de um **array** quando este array é destruído. Isto é feito implicitamente para arrays que não são alocados utilizando-se *new*. No entanto, isso não pode ser feito implicitamente para **arrays** alocados na memória dinâmica porque o compilador não consegue determinar se um ponteiro aponta para um objeto único ou para o primeiro elemento do array de objetos. Por exemplo:

```
void f()
{
        BigInt* a = new BigInt;
        BigInt* b = new BigInt[10];
        delete a;       // OK
        delete b;       // Problemas!!!
}
```

Neste caso deve-se especificar que *b* aponta para um **array**:

```
void f()
{
        BigInt* a = new BigInt;
        BigInt* b = new BigInt[10];
        delete a;               // OK
        delete[10] b;           // OK!!!
}
```

Desta forma, sabe-se quantas instâncias de classe o array contém, podendo assim chamar o destrutor para cada instância do array.

3.2.16 Funções Inline

A utilização de funções membros pequenas é comum quando se programa orientado a objetos. O emprego de tais funções, pode gerar uma considerável ineficiência devido ao custo de uma chamada de função.

Sendo assim, C++ permite que se declare uma função como *inline*. Nesse caso, cada chamada de função é substituída por uma cópia da função inteira, semelhante a expansão de macro. Isso elimina o custo de uma chamada de função.

Há duas maneiras de se definir uma função como sendo inline:

• funções *inline* definidas dentro da especificação da classe - Quando uma função membro é definida dentro da própria classe, o compilador faz uma chamada em linha dessa função toda vez que o compilador a chamar. Por exemplo, o destrutor da classe *BigInt*.

• funções inline definidas fora da especificação da classe - Para que uma função membro seja *inline* sem ser definida dentro da classe, deve-se colocar a palavra reservada *inline* antes da definição da função.

Deve-se recomendar, contudo, um certo cuidado ao se utilizar funções *inline*. As funções candidatas a serem colocadas em linha são as que apresentam pouco código e este código também é bastante simples.

Capítulo IV - Conceitos de Programação Orientada a Objetos em C++

4.1 Introdução

As características mais interessantes da linguagem C++ são aquelas que suportam o estilo de Programação Orientada a Objetos (POO). Programação orientada a objetos é uma técnica de organização de tipos abstratos de dados que explora as características em comum destes tipos de modo a reduzir o esforço de programação através da reutilização de código. Esse estilo de programação se caracteriza por:

- Encapsulamento
- Herança
- Amarração Dinâmica

Nos capítulos anteriores discutiu-se encapsulamento, que restringe os programas clientes a interagirem com as instâncias da classe somente por meio de um conjunto bem definido de operações: a interface (especificação) da classe. Isso evita que os programas clientes conheçam detalhes de como uma classe representa os dados que ela contém e dos algoritmos que ela utiliza, ou seja, da implementação de uma classe. Isso faz com que as mudanças (corretas) que ocorram na implementação de uma classe não tenham nenhum efeito permissivo nos programas clientes desta classe, além de facilitar o processo de depuração e manutenção porque o código que lida com os dados de uma classe está localizado na própria classe.

Herança simplifica a tarefa de criação de uma nova classe que seja similar a uma classe existente, uma vez que permite ao programador expressar apenas as diferenças existentes entre a nova classe e a classe existente, ao invés de exigir que o programador crie uma classe a partir do começo.

Amarração Dinâmica, ou amarração tardia, ajuda a fazer um programa cliente mais geral, ocultando as diferenças entre um grupo de classes que estão relacionadas entre si. Amarração Dinâmica permite que cada classe de um grupo de classes relacionadas tenha uma implementação diferente para uma função particular. Programas clientes podem aplicar a operação a uma instância de classe sem ter que considerar a classe específica da instância. Em tempo de execução, determina-se a classe específica da instância e chama-se a implementação da função para esta classe.

Sendo assim, encapsulamento, herança e amarração dinâmica trabalham juntas:

- Encapsulamento assegura que os programas cliente interagem com os objetos somente por meio das funções membros.

• Herança proporciona um meio de se expressar as diferenças entre as classes, mas permitindo o reuso do código e das variáveis membros para implementar as características similares entre elas.

• Amarração Dinâmica das funções usadas pelos programas clientes oculta as diferenças entre as classes expressas via herança.

Foi visto anteriormente que C++ suporta encapsulamento através da definição dos atributos públicos e privados de uma classe. C++ suporta herança por meio de classes derivadas e suporta amarração dinâmica por meio de funções virtuais.

O presente capítulo introduzirá as características de C++ que suportam o estilo de programação orientada a objetos, em especial herança e amarração dinâmica.

4.1.1 Classes Derivadas

O mecanismo de herança (B herda de A) associa à classe que está sendo declarada (B):

• Uma representação inicial para os seus objetos (aquelas dos objetos de A);
• Um conjunto inicial de métodos aplicáveis aos objetos desta classe (aqueles da classe A).

A nova declaração de classe pode conter representação e operações adicionais, especializando estado e o comportamento de novos objetos a serem criados. Deve-se notar, contudo, que os métodos iniciais são ainda aplicáveis aos objetos desta nova classe.

Esta visibilidade tem relação com o conceito de subtipo. Suponha uma classe *Pessoa* em C++, como declarada abaixo:

```
class Pessoa
{
        char* nome;
        int idade;
public
        Pessoa ( char*auxnome, int a) {nome=auxnome; idade=a;}
         void MudarIdade (int a) {idade=a;}
         virtual void Print ( );
 };
```

Suponha também que se necessite representar empregados em um programa. Uma vez que todo o empregado é basicamente uma pessoa pode-se tomar vantagem da declaração existente para pessoa, declarando-se uma nova classe *Empregado* derivada da classe *Pessoa*:

```
class Empregado: public Pessoa
```

{
};

Com esta declaração pode-se criar instâncias de *Empregado* (objetos):

Empregado e1, e2;

Contudo, nada foi modificado até agora, uma vez que empregados são exatamente iguais a pessoas, tanto na estrutura de dados quanto no comportamento. Isso se deve ao fato de que nenhuma declaração adicional foi acrescentada localmente a *Empregado*. Obviamente faz-se necessário refinar a classe *Pessoa* de maneira a representar um empregado. Isto pode ser feito adicionando-se novas estruturas de dados e operações dentro da classe *Empregado*, como mostrado abaixo:

```
class Empregado : public Pessoa {
        float salario;
public:
        Empregado(char *auxnome, int auidade, float auxsal) : Pessoa (auxnome, auxidade)  { salario=auxsal;}
        void MudarSalario (float r)     { salario = r; }
        void Print ();
};
```

Com essa nova declaração da classe *Empregado*, o mecanismo de herança estabelece que:

- todo empregado tem a seguinte estrutura de dados: *nome*, *idade* e *salario*.
- todo empregado pode receber as seguintes operações: *MudarIdade*, *MudarSalario*, *Print*.

Pode-se notar que uma operação para mudar a idade de um empregado não necessita ser redeclarada na classe *Empregado*; o método *MudarIdade* na classe *Pessoa* serve perfeitamente para mudar a idade de um empregado. Sendo assim, pode-se escrever:

e1.MudarIdade (56);

mudando a idade de *e1* para 56 anos.

Uma classe derivada herda todos os atributos (variáveis membros e funções) da classe base. Pode-se, contudo, diferenciar a classe derivada de sua classe base de três formas:

- adicionando variáveis membros;
- adicionando funções membros;
- redeclarando funções membros herdadas da classe base.

Uma classe base pode ter mais de uma classe derivada, uma classe derivada pode, por sua vez, servir como classe base para outras classes derivadas. Uma classe derivada normalmente é mais especializada que sua classe base.

4.1.2 Redeclaração - Funções Membros Virtuais

Uma outra característica relacionada com o mecanismo de herança é a habilidade de redeclarar operações na subclasse. Através da redeclaração nós objetivamos a criação, na subclasse, de uma nova operação com a mesma interface (nome, parâmetros e resultado) que uma operação na superclasse. Se uma operação da superclasse é redeclarada em uma subclasse, então a nova operação é visível para os usuários da subclasse. Contudo, a operação da superclasse é também aplicável dentro da subclasse desde que o nome da subclasse preceda o nome da operação (em caso contrário, uma chamada recursiva ocorrerá) . A operação redeclarada tem a habilidade de manipular a representação de dados inteira da subclasse, chamando a operação homônima da superclasse para lidar com a representação comum, e incluindo declarações para lidar com a representação adicional. Por exemplo:

```
void Empregado::Print ( )
{
        Pessoa :: Print ( );
        cout « salario « "\n";
}
```

A palavra chave *virtual* em C++ indica que a função *Print* da classe *Pessoa* pode ter diferentes versões para diferentes classe derivadas. Quando um programa cliente aplica uma função virtual a um objeto, C++ automaticamente determina a que classe o objeto pertence, em tempo de execução, e transfere o controle para a implementação correta da função. Isso ocorre mesmo quando o programa cliente trata o objeto como uma instância da classe base. Deve-se ressaltar que uma função virtual deve ser definida na primeira classe onde ela é declarada.

Uma classe base com múltiplas classes derivadas, além de permitir a eliminação de código redundante, também cria uma oportunidade de se fazer programa clientes mais gerais.

4.2 Um Exemplo de Aplicação

Nesta seção será desenvolvido um programa gráfico simples que desenha algumas figuras geométricas em duas dimensões. Este programa utilizará os conceitos de encapsulamento, herança e amarração dinâmica.

4.2.1 Classe Ponto

Esse exemplo utiliza um tipo abstrato de dados, chamado de classe *Ponto*, para executar operações sob pares de coordenadas (x, y):

```
// Hierarquia de classes geometria

class Ponto{
        int xc, yc;                     // coordenadas x, y
public:
        Ponto ( )                       {xc = yc = 0;}
```

```
        Ponto (int novox, int novoy)        {xc = novox; yc = novoy}
        int x ( ) const                     {return xc;}
        int x (int novox)                   {return xc = novox;}
        int y ( ) const                     {return xy;}
        int y (int novoy)                   {return yc = novoy;}
        Ponto operator+ (const Ponto&p) const {
                return Ponto (xc + p.xc, yc + p.yc);
        }
        void operator+= (const Ponto & p) {
                xc += p.x ( );
                yc += p.y ( );
        }
        void printOn() const
        {
                cout << "( " << xc << ", " << yc << " )";
        }
}

ostream& operator<< (ostream& strm, const Ponto& p)
{
        return strm << "( " << p.x() << ", " << p.y() << " )";
}
```

A classe *Ponto* não utiliza nenhuma característica de C++ que não tenha sido discutida anteriormente. Para que se possa utilizar o operador << para objetos da classe *Ponto* é necessário sobrecarregá-lo. Maiores detalhes de como sobrecarregar o operador << serão discutidos posteriormente.

4.2.2 Classe Linha e Classe Circulo

O exemplo geométrico requer algumas classes para representar várias formas geométricas, como linha, retângulo, triângulo e círculo. Todas essas classes têm algumas operações em comum, como por exemplo, desenhar( ) e mover( ), e todas têm uma variável membro, *org*, que contém as coordenadas de origem da forma geométrica. Pode-se começar escrevendo-se declarações representativas para as classes *Linha* e *Circulo*, como por exemplo:

```
class Linha{
        Ponto org;              // origem
        Ponto p;                // ponto de fim
public:

        Linha (const Ponto & a, const Ponto & b): org (a.x(),a.y()), p(b.x(),b.y()) {}
        void mover (const Ponto & d);           // move linha de um montante d
        void desenhar() const ;                 // desenha uma linha de org ate p
```

```
};


class Circulo {
        Ponto org;                      // origem
        int raio;                       // raio do circulo
public:
        Circulo(const Ponto & c, int r): org (c.x(),c.y()) { raio = r;}
        void mover(const Ponto & d);    // move um circulo com centro org
        void desenhar( ) const;         // desenha um circulo com centro org e raio r
};
```

4.2.3 Instâncias de Classe como Variáveis Membros de uma Classe

Pode-se ver, dos exemplos acima, que uma instância de uma classe pode ser uma variável membro de outra classe: as variáveis membro *Linha::org*, *Linha::p*, *Circulo::org* são instâncias da classe Ponto. Quando isto ocorre, o construtor da classe pode necessitar utilizar uma notação especial, para inicializar as variáveis membros que aparecem como instâncias de uma classe. Entre a lista de argumentos da função e o corpo da função, escreve-se o nome das variáveis membros que se deseja inicializar, seguido por uma lista de argumentos que se deseja passar para os construtores das variáveis membros:

```
Linha (const Ponto& a, const Ponto& b ): org (a.x(),a.y()), p (b.x(),b.y()) {}
Circulo (const Ponto& c, int r): org (c.x(),c.y()) {raio = r;}
```

O construtor de cada uma das variáveis membros é processado antes que as instruções no corpo do construtor da classe seja executado.

Pode-se também escrever:

```
Circulo (const Ponto & c, int r)
{
        org = c;
        raio = r;
}
```

mas é provável que isto não seja tão eficiente. Relembre que, se uma classe declara um ou mais construtores, então C++ garante que um deles será chamado para inicializar toda instância da classe. Relembre também que a atribuição *org = c* assume que *org* já está inicializado.

4.2.4 A Função Membro mover()

As funções membros *Linha::mover()* e *Circulo::mover()* aceitam um *Ponto* como argumento, representando um deslocamento (x, y), e então os adiciona as variáveis membros:

```
void Linha::mover(const Ponto& d)
{
        org += d;
         p += d;
}

void Circulo::mover(const Ponto& d)
{
            org += d;
}
```

4.2.5 A Função Membro desenhar()

As implementações das funções membros das classes *Linha* e *Circulos* estão completas a menos para as funções *Linha::desenhar* e *Circulo::desenhar().* Afim de manter estes exemplos simples, os detalhes de como se desenhar estas formas em um dispositivo gráfico particular será omitido, sendo assim a implementação apenas imprimirá o tipo da forma e suas coordenadas:

```
void Linha::desenhar() const
{
        cout << "Linha de " << org << "até" << p << "\n";
}

void Circulo::desenhar() const
{
        cout << "Circulo com centro em " << org << " e raio de " << raio << "\n";
}
```

Seria útil ter um tipo abstrato de dados chamado de *Quadro* que seria uma coleção de *Linhas*, *Triangulos*, *Retangulos* e *Circulos*. Também seria útil ser capaz de *desenhar()* e *mover()* os quadros.

Seria extremamente elegante se a classe *Quadro* fosse geral, e não contivesse nenhuma menção a formas específicas. Desta maneira, poderia se introduzir um nova forma, por exemplo trapézio, sem que haja necessidade de se alterar a classe *Quadro*. Pode-se fazer isto definindo-se um classe base *Forma* com classes derivadas *Linha*, *Triangulo*, e assim sucessivamente, como mostrado na figura 5.

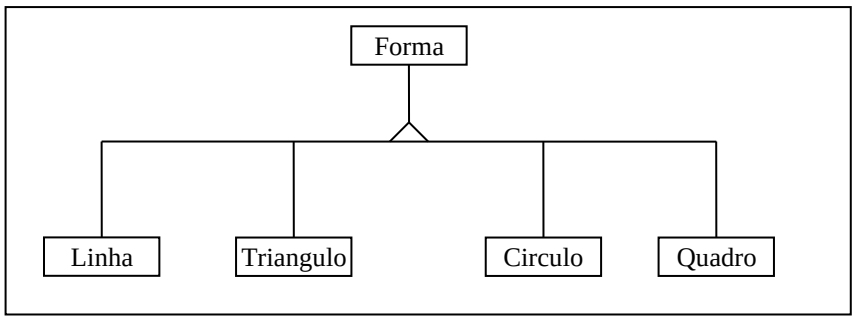


Figura 5 - Hierarquia de Exemplo

A primeira versão da classe *Forma* declara funções, tais como *desenhar()* e *mover()*, como sendo funções virtuais. Estas funções podem ser aplicadas a qualquer tipo de forma e, são implementadas para que escrevam uma mensagem de erro se chamadas:

```
class Forma {
public:
        virtual void mover (const Ponto&);
        virtual void desenhar() const;
};

void Forma::mover (const Ponto& p)
{
        cout << "Esqueceu de implementar mover()! \n";
        exit (1);
}

void Forma::desenhar () const
{
        cout << "Esqueceu de implementar desenhar()! \n";
        exit (1);
}
```

Deve-se mudar as declarações da classe *Linha*, *Circulo* e, assim sucessivamente para fazê-las classes derivadas (herdeiras) da classe *Forma*, isto é feito adicionando-se o nome da classe base *Forma*, às declarações das classes derivadas:

```
class Linha: public Forma {
        Ponto org;                              // origem
        Ponto p;                                // ponto de fim
public:
        Linha (const Ponto& a, const Ponto& b): org (a.x(), a.y()), p(b.x(), b.y()) {}
        virtual void mover (const Ponto& d);    // mover linha um montante d
        virtual void desenhar () const;         // desenha uma linha de org ate p
};



class Circulo: public Forma {
        Ponto org;                              // origem
        int raio;                               // raio do circulo
public:
        Circulo (const Ponto& c, int r): org(c.x(), c.y())  { raio = r; }
        virtual void mover (const Ponto& d);    // move um circulo de um montante d
        virtual void desenhar() const;          // desenha um circulo com centro org
                                                // e raio  raio
};
```

Utilizando-se da palavra reservada ***public*** (especificador de acessabilidade) antes do nome da classe base torna os membros públicos da classe base membros públicos da classe derivada, de modo que os clientes da classe derivada podem ter acesso a eles. A omissão da palavra reservada ***public*** torna os membros públicos da classe base membros privados da classe derivada, e, portanto, inacessíveis aos clientes da classe derivada.

A palavra reservada ***virtual*** também é adicionada às declarações das funções *desenhar()* e *mover()* nas classes derivadas, mas não há motivo para mudar a implementação destas funções[3].

4.2.6 Classe Quadro

Pode-se agora escrever a classe *Quadro* para lidar somente com a classe *Forma* e não suas classes derivadas *Linha*, *Triangulo*, *Retangulo* e *Circulo*, etc. Pode-se representar um *Quadro* como um **array** contendo ponteiros para as suas formas componentes. Pode-se, também, implementar *Quadro::desenhar()*, por exemplo, simplesmente chamando-se *desenhar()* para cada forma no quadro:

```
const unsigned CAPACIDADE_QUADRO = 100;

class Quadro: public Forma {
        Ponto org;                              // origem
        Forma* s[CAPACIDADE_QUADRO];            // ponteiro de array para formas
        int n;                                  // numero de formas neste quadro
public:
        Quadro() {n = 0;}                       // construtor
        Quadro(Ponto& o): org (o.x(), o.y())    {n = 0;}        // construtor
        void adicionar (Forma&);                // adicionar forma a quadro
        virtual void mover (const Ponto& d);    // mover quadro
        virtual void desenhar() const;          // desenhar quadro
};

void Quadro::adicionar(Forma & t)
{
        if (n == CAPACIDADE_QUADRO) {
                cerr <<  "Capacidade do quadro excedida\n";
                exit(1);
        }
        s[n++] = &t;                    // adicionar ponteiro para forma a quadro
}

void Quadro::mover(const Ponto& d)
{
        org += d;
```

[3] A palavra reservada virtual é requerida somente nas declarações das funções na classe base. Pode-se omitir a palavra virtual nas classes derivadas - contudo considera-se um bom estilo de programação, porque torna possível identificar quais funções são virtuais sem ter que olhar as declarações na classe base.

```
        for (int i = 0; i < n; i++) s[i]->mover(d);
}

void Quadro::desenhar() const        // desenhar um Quadro
{
        for (int i = 0; i<n; i++) s[i]->desenhar();
}
```

Uma vez que *desenhar()* é uma função virtual, C++ tem o cuidado de determinar a classe específica de cada componente *Forma* quando o programa é executado e então chama a implementação apropriada de *desenhar()* para cada classe. Isto é chamado de amarração dinâmica, em contraste com a chamada de funções não-virtuais, as quais empregam amarração estática. Por exemplo, se em tempo de execução *s[i]* aponta para uma instância da classe *Linha*, então a instrução *s[i]->desenhar()* chama *Linha::desenhar()*; se *s[i]* aponta para uma instância de um *Circulo*, chama-se *Circulo::desenhar()*, e assim sucessivamente. Considere o seguinte programa exemplo:

```
main()
{
        Linha l(Ponto(1,2), Ponto(3,4));     // cria uma linha
        Circulo c(Ponto(5,6),1);             // cria um circulo
        l.desenhar();                                // desenha uma linha
        c.desenhar();                        // desenha um circulo
        Quadro q;                            //cria um quadro vazio
        l.mover(Ponto(1,0));                 // move a linha
        q.adicionar(l);                      // adiciona a linha ao quadro
        c.mover(Ponto(1,0));                 // move o circulo
        q.adicionar(c);                      // adiciona o circulo ao quadro
        q.desenhar();                        // desenha o quadro
        q.mover(Ponto(10,10));               // translada o quadro para (10,10)
        q.desenhar();                        // desenha o quadro novamente
}
```

Quando se executa este programa, o resultado obtido é:

```
Linha de (1,2) ate (3,4)
Circulo com centro em (5,6) e raio 1
Linha de (2,2) ate (4,4)
Circulo com centro em (6,6) e raio 1
Linha de (12,12) ate (14,14)
Circulo com centro em (16,16) e raio 1
```

Amarração dinâmica de uma chamada de função virtual ocorre somente quando a função virtual é aplicada a um ponteiro ou referência a um objeto, como em *Quadro::desenhar()*, e não quando é aplicada diretamente a um objeto:

```
void desenharIsto(Forma& referencia)
{
        Linha linha(Ponto(1,2), Ponto(3,4));
```

```
    Forma* ponteiro = &linha;
    //...
    ponteiro->desenhar();          // amarrado dinamicamente
    referencia.desenhar();         // amarrado dinamicamente
    linha.desenhar();              // amarrado estaticamente
}
```

Em tempo de execução, as variáveis ponteiro e referência podem apontar para uma instância de qualquer classe derivada de *Forma*. Sendo assim, C++ utiliza amarração dinâmica para determinar qual implementação de *desenhar* deve chamar. No exemplo, a variável *linha* é uma instância da classe *Linha*, portanto, C++ estaticamente amarra a chamada de *linha.desenhar()* a *Linha::desenhar()* e evita a sobrecarga da amarração dinâmica.

4.2.7 Compatibilidade de Tipos

Porque o compilador C++ não acusa erro de compatibilidade de tipos nas seguintes chamadas de funções: *q.adicionar(l)* e *q.adicionar(c)*, uma vez que a declaração para *Quadro.adicionar()* especifica que esta função requer um argumento do tipo *Forma&*, e o argumento passado para estas duas chamadas de funções são do tipo *Linha e Circulo*, respectivamente. C++ permite isto porque *Linha* e *Circulo* são classes derivadas da classe *Forma*. Em geral, C++ permite que uma referência ou ponteiro para uma instância de uma classe derivada seja utilizada em lugar de uma referência ou ponteiro para a sua classe base sem que seja necessário uma conversão de tipos explícita (**cast**). Pode-se, portanto, entender uma classe derivada como sendo um subtipo de sua classe base. Subtipos são úteis e não comprometem a segurança porque classes derivadas herdam todas as variáveis membros e funções de sua classe base; sendo assim, uma referência a qualquer variável membro ou chamada de função membro de uma instância da classe base está bem definida para uma instância da classe derivada. Pode-se pensar em *Linha* e *Circulo* como sendo espécies diferentes (subtipos) de *Formas*.

O oposto, contudo, não é verdade: C++ não permite que uma referência ou um ponteiro para uma instância de uma classe base seja utilizada em lugar de referência ou ponteiro para uma instância de uma classe derivada. Isto reflete o fato de que uma *Linha* ou *Circulo* é uma espécie de *Forma*, mas uma *Forma* não é um tipo de *Linha* ou *Circulo*.

4.2.8 Classe Incompletas - Funções Virtuais Puras

Classes também podem ser incompletas. Isto ocorre quando uma ou mais operações são declaradas em uma classe, mas não são implementadas. A implementação destas operações é deixada para as classes dependentes. Isto é conveniente para disciplinar a construção de classes: para cada operação incompleta em uma superclasse, todas as subclasses herdeiras desta superclasse devem implementar ou virtualizar esta operação. Classes incompletas não modelam objetos, mesmo se uma ou mais de suas operações está completamente declarada.

Afim de exemplificar a conveniência das operações virtuais puras, considere novamente a classe *Forma* descrita anteriormente. Pode-se transformar esta classe em uma classe incompleta da seguinte forma:

```
class Forma {
public:
        virtual void mover(const Ponto&) =0;
        virtual void desenhar() const =0;
};
```

O inicializador *=0* nas declarações de *mover()* e *desenhar()* indicam ao compilador C++ que estas são funções virtuais puras: funções virtuais que a classe base não implementa, e que, portanto, devem ser definidas pelas classes derivadas. Há duas consequências de se declarar um função virtual pura na classe base:

• O compilador verifica cada classe derivada para assegurar que elas implementam a função virtual pura da classe base ou adiam sua implementação (declarando-a novamente como função virtual pura a ser implementada pelas suas classes derivadas).

• Não se pode criar instâncias de uma classe incompleta. Sendo assim, a seguinte declaração é ilegal:

```
Forma f;
```

As vantagens da utilização de declarações incompletas podem ser resumidas da seguinte forma:

- Melhoria da organização hierárquica, através do encapsulamento de propriedades na raiz da estrutura;
- Uma maior disciplina na programação, já que força o comportamento necessário nas classes descendentes;
- Incentiva o uso de polimorfismo, permitindo um comportamento mais abstrato e genérico para os objetos.

Deve-se ressaltar que nem todas as classes bases são classes abstratas. Por exemplo, pode-se implementar uma classe desenhando arcos com uma classe derivada da classe Circulo:

```
class Arco : public Circulo {
        float inicio, fim;              // angulo de inicio e de fim do arco
public:
        Arco(const Ponto& c, int r, float a1, float a2) : Circulo(c,r) {
                inicio = a1;
                fim = a2;
        }
        virtual void desenhar();
};
```

Instâncias da classe *Circulo* são ainda úteis, apesar de se utilizar esta classe como classe base da classe *Arco*.

O desenvolvimento da classe *Arco* também ilustra como uma hierarquia de classes pode crescer através do processo de especialização das classes existentes.

4.3 O Exemplo Geométrico Melhorado

O próximo passo no desenvolvimento de nosso exemplo de programa gráfico consiste em tirar proveito do fato de que todas as classes derivadas de *Forma* possuem uma variável membro comum, *org*. Sendo assim, pode-se mover esta variável membro das classes derivadas para a classe base *Forma*. A função membro *Forma::mover()* pode agora ajustar *org*, logo ela não é mais uma função virtual pura. A classe base *Forma* agora é a seguinte:

```
class Forma {
        Ponto org;                                              // origem
public:
        Forma(const Ponto& p): org(p.x(), p.y())    {}
        Ponto origem() const                    { return org; }
        virtual void mover(const Ponto& d)  { org += d;}
        virtual void desenhar() const =0;
};
```

Uma vez que a classe Forma tem agora uma variável membro, pode-se prover um construtor para inicializá-la e a função membro *origem()* para ler o seu valor.

Utilizando-se esta definição da classe *Forma* pode-se simplificar as classes derivadas *Linha*, *Circulo* e *Quadro*, uma vez que elas herdam as funções membros *origem(), mover(), desenhar()* e a variável *org* da classe *Forma*:

```
class Linha: public Forma {
        Ponto p;                                // ponto de fim
public:
        Linha (const Ponto& a, const Ponto& b): Forma(a), p(b.x(), b.y())       {}
        virtual void mover(const Ponto&);               // mover linha de um montante d
        virtual void desenhar() const;                  // desenha uma linha de org ate p
};

void Linha::mover(const Ponto& d)
{
        Forma::mover(d);
        p+= d;
}

void Linha::desenhar() const
```

```
{
        cout << " Linha de " << origem() << "ate " << p << "\n";
}

classe Circulo: public Forma {
        int raio;                                   // raio do circulo
public:
        Circulo(const Ponto& c, int r): Forma (c)   { raio = r }
        virtual void desenhar() const;              // desenha um circulo com
                                                    // centro org e raio raio
};

void Circulo::desenhar() const
{
        cout << " Circulo com centro em " << origem() << " e raio de " << raio
<< "\n";
}

const unsigned CAPACIDADE_QUADRO = 100;

class Quadro: public Forma {
        Forma* s[CAPACIDADE_QUADRO];        // array de ponteiros para formas
        int n;                                      // numero de formas neste quadro
public:
        Quadro(): Forma(Ponto(0,0))         { n = 0; }      // construtor
        Quadro(Ponto& org): Forma (org)  { n = 0;}      // construtor
        void adicionar(Forma&);                     // adiciona Forma a Quadro
        virtual void mover(const Ponto&);           // mover quadro de um montante d
        virtual void desenhar() const;                      // desenhar quadro;
};




void Quadro::adicionar(Forma& t)
{
        if (n == CAPACIDADE_QUADRO) {
                cerr << " Capacidade do quadro excedida \n" ;
                exit(1);
        }
        s[n++] = &t;                    // adiciona ponteiro para Forma a Quadro
}

void Quadro::mover(const Ponto& d)
{
        Forma::mover(d);
        for (int i = 0; i < n; i++) s[i]->mover(d);
}

void Quadro::desenhar() const
```

```
{
        for (int i=0; i<n; i++) s[i]->desenhar();
}
```

Deve-se observar que enquanto a classe *Circulo* pode simplesmente herdar *Forma::mover(),* a classe *Linha* não pode uma vez que esta deve também ajustar o outro ponto extremo *p* quando da operação *mover*.

4.3.1 Inicialização de Objetos

Os construtores da classe *Linha*, *Circulo* e *Quadro* ilustram como uma classe derivada pode passar argumentos para um construtor de sua classe base. Para tal, deve-se utilizar a seguinte sintaxe:

Nome_Classe_Derivada (parâmetros da classe base, parâmetros da classe derivada[4]): Nome_Classe_Base(parâmetros da classe base), Construtores_Membros (parâmetros para construtores dos membros)
{
        //...
        definição da função
        //...
}

Por exemplo:

```
Linha (const Ponto& a, const Ponto&b) : org(a.x(), a.y()), p(b.x(), b.y())  {}
Circulo (const Ponto& c,int r) : Forma (c)          { raio = r }
Quadro (): Forma (Ponto(0,0))                {n = 0 ;}
Quadro (Ponto& org): Forma (org)             {n = 0 ;}
```

As regras para se determinar a ordem pela qual C++ inicializa as instâncias da classe base e variáveis membros em um objeto são um pouco complexas. A ordem de inicialização no caso onde as classes têm, no máximo, uma única classe base é a seguinte:

1. a classe base, se existir alguma;
2. as variáveis membros, na ordem em que são declaradas na declaração de classe, *não* na ordem em que elas aparecem na lista de parâmetros do construtor.

O que complica é o fato de que C++ aplica estas regras recursivamente; por exemplo, se a classe base tem uma classe base, C++ inicializa a classe base da classe base e qualquer membro que esta possa conter antes de inicializar a classe base. O exemplo seguinte mostra a ordem pela qual os construtores são chamados durante a inicialização de um objeto complexo:

```
#include <iostream.h>

class X {
        int i;
```

[4] A ordem dos parâmetros na realidade não importa.

```
public:
        X(const char* s)        { cout << s << ' ';}
        X()                     { cout << "X::X() ";}
};

class A {
        X a1;
        X a2;
public:
        A(const char* s) : a2("A::a2")        {cout << s << ' ';}
};

class B: public A{
        X b1;
        X b2;
public:
        B(const char* s): b2("B::b2"), b1("B::b1"), A("B::A")    {cout << s << ' ';}
};

int initCi()
{
        cout << "C::i";
        return 0;
}

int& initCr()
{
        static int n = 1;
        cout << "C::r";
        return n;
}

class C: public B {
        cont int i;
        int& r;
        X c1;
        X c2;
public:
        C(const char* s): B("C::B"), c1("C::c1"), r(initCr()), i (initCi()),
                          c2("C::c2")         {  cout << s << "\n"; }
};

main()
{
        C c("c");
}
```

A saída deste programa é :

```
X::X() A::a2 B::A B::b1 B::b2 C::B C::i C::r C::c1 C::c2 c
```

Tente demonstrar como as regras produzem estes resultados.

Este exemplo também mostra como inicializar variáveis membros constantes e referências (*C::i* e *C::r* neste caso) - eles devem ser inicializados pela lista de inicialização do construtor.

4.3.2 Finalização de Objetos

A regra que C++ utiliza para determinar a ordem pela qual destrutores são chamados é simples: destrutores são chamados na ordem inversa dos construtores. Isto explica porque C++ ignora a ordem dos membros na lista de inicialização e utiliza a ordem dos membros na declaração de classe: uma classe pode ter vários construtores, e cada um destes construtores poderia ordenar os membros em sua lista de inicialização de uma maneira diferente. No momento de destruir um objeto, o destrutor não saber qual construtor foi utilizado para criar o objeto. Utilizando-se a ordem dos membros na declaração de classe evita-se este problema, permitindo-se que os destrutores das variáveis membros sejam chamados na ordem inversa, sem ter que considerar qual construtor criou o objeto.

Capítulo V - Conceitos Complementares de Programação em C++

5.1 Introdução

Agora que você já conhece os principais conceitos de programação na linguagem C++, vamos apresentar alguns conceitos adicionais desta linguagem de programação. Embora estes conceitos não sejam parte do núcleo básico da linguagem, os tópicos abordados neste capítulo final definem características complementares de C++ que facilitam o entendimento de programas e permitem a construção de programas mais flexíveis.

5.2 Construtor Pré-Definido de Cópia, Operador Pré-definido de Atribuição e Construtor Pré-Definido Default

Ao se construir uma nova classe, é importante identificar se nenhuma das variáveis membro têm, como seus valores, objetos dinâmicos (isto é, objetos criados através do uso de alocação dinâmica). Se for este o caso, não é necessário definir um construtor de cópia e um operador de atribuição específicos para esta classe, visto que C++ pré-define um construtor de cópia e um operador de atribuição que é válido para todas as classes.

O construtor pré-definido de cópia realiza a cópia membro a membro das variáveis do objeto que será copiado para as variáveis do objeto que será a cópia. Da mesma maneira, o operador pré-definido de atribuição realiza a atribuição membro a membro das variáveis do objeto que será atribuído para as variáveis do objeto que recebe a atribuição.

Se a classe não envolve objetos dinâmicos, pode-se garantir que a cópia e a atribuição de objetos será feita de maneira apropriada. O mesmo não pode ser dito quando a classe envolve objetos dinâmicos. Neste caso, a utilização do construtor e do operador pré-definido pode gerar sinonímias e efeitos colaterais. Recomenda-se neste caso, portanto, que sejam definidos um construtor de cópia e um operador de atribuição específicos para a classe em questão.

Por exemplo, a classe *Ponto*, apresentada no capítulo IV, não requer que sejam definidos o construtor de cópia e o operador de atribuição, visto que suas variáveis membros *xc* e *yc* são do tipo inteiro. Por outro lado, a classe *BigInt*, requer que eles sejam definidos porque a variável membro *digitos* tem como valor um ponteiro para um vetor de dígitos alocado dinâmicamente.

Ao se construir uma classe é possível deixar de especificar construtores para ela. Isto ocorre porque, quando **não** existem construtores para uma classe, o

compilador criará um construtor default para ela. Deste modo, o código seguinte funciona;

```
//: C06:AutoDefaultConstructor.cpp
// construtor default gerado automaticamente
class V {
        int i; // private
}; // sem construtor
int main() {
        V v, v2[10];
} ///:~
```

Contudo, se ao menos um construtor da classe for definido, o compilador não criará o construtor default e as instâncias de *V* acima gerariam erros em tempo de compilação. É importante tomar cuidado com o uso do construtor default pré-definido porque ele não assegura nenhum tipo de ação especial (por exemplo, inicializar as variáveis membro numéricas com zero). Portanto, se você quer inicializar variáveis membro com zero, você deve fazer isso por sua própria conta escrevendo o construtor default explicitamente.

Como vimos nesta seção, o compilador C++ pode fornecer construtores pré-definidos e operadores de atribuição pré-definidos para as suas classes. Contudo, muitas vezes, o comportamento destes construtores e operadores não são os que você deseja. Deste modo, esta característica de C++ deve ser encarada como uma rede de segurança, mas não deve ser amplamente utilizada. Em geral, trata-se de boa política definir os contrutores explicitamente e não deixar que o compilador faça isto por você.

5.3 Objetos Temporários

Em algumas situações, é necessário ou conveniente que o compilador de C++ crie objetos temporários. A criação destes objetos é dependente da implementação do compilador. Sempre que um objeto temporário é criado, o construtor desta classe deve ser chamado. Da mesma maneira, o destrutor da classe deve ser chamado quando o objeto temporário é eliminado. Por exemplo,

```
class X {
        // ...
public:
        X(int);
        X(const X&);
        ~X();
};
X f(X);
void g() {
        X b = f(X(2));
}
```

Um objeto temporário pode ser criado para abrigar o resultado da chamada ao construtor *X(2)*. Posteriormente, este objeto será passado como argumento para *f*.

Uma vez que o objeto tenha sido passado como argumento, o destrutor já pode ser chamado.

5.4 Retorno de Objetos em Funções

Objetos podem ser retornados por funções. Sempre que isto ocorre, o construtor de cópia (definido pelo usuário ou o pré-definido) é chamado para criar o objeto que será retornado. Pode-se usar no comando *return* objetos cujo tipo podem ser convertidos (de forma pré-definida ou através do uso de construtores) para a classe do objeto que deve ser retornado. Por exemplo, na função *f* abaixo, o valor *0* será implicitamente convertido num *BigInt*, antes de ser retornado, através do uso de um construtor da classe *BigInt*.

```
BigInt f() { return 0; }
```

Objetos também podem ser retornados através do uso de ponteiros e referências para o objeto. Nestes casos, é importante tomar cuidado para não retornar ponteiros e referências para variáveis locais, como acontece nos exemplos a seguir:

```
Ponto* f () {
        Ponto a;
         //...
        return &a;                  // erro
}

Ponto& f () {
        Ponto a;
         //...
        return a;                   // erro
}
```

O retorno de referências objetiva permitir o uso de funções no lado esquerdo de atribuições, no acesso seletivo de variáveis compostas ou em atribuições múltiplas. Como o operador de atribuição da classe *BigInt* retorna *void*, não é permitido fazer atribuições múltiplas de *BigInt*s. Logo, o código a seguir é incorreto:

```
BigInt c = 243;
BigInt a, b;
a = b = c;                      // erro
```

Para que a atribuição acima seja permitida, é necessário alterar a declaração e definição do operador atribuição de modo a retonar uma referência a *BigInt*, como mostrado abaixo:

```
BigInt& BigInt::operator=(const BigInt& n)
{
        if (this == &n) return *this;      // para manipular x = x corretamente
```

```
        delete digitos;
        unsigned i = n.digitos;
        digitos = new char[n.digitos=i];
        char* p = digitos;
        char* q = n.digitos;
        while (i--) *p++ = *q++;
        return *this;
}
```

5.5 Especificadores de Acesso

Como já tivemos a oportunidade de ver, C++ permite que se controle o acesso a variáveis e funções membros de uma classe através dos especificadores *public*, *private* e *protected*. Quando um membro é *private*, seu nome pode ser apenas usado pelas funções membro e *friend*s da classe na qual ele é declarado. Quando ele é *public*, seu nome pode ser usado por qualquer função. Quando ele é *protected*, seu nome só pode ser usado or funções membro e *friend*s da classe na qual ele é declarado, e por funções membro e *friend*s de classes derivadas desta classe.

C++ também permite que se especifique se uma classe herda os membros da classe base de maneira pública ou privada. Quando se usa o especificador de acesso *public*, os membros públicos da classe base são membros públicos da classe derivada e os membros protegidos da classe base são membros protegidos da classe derivada. Quando se usa o especificador de acesso *private*, os membros públicos e protegidos da classe base são membros privados da classe derivada. Se o especificador de acesso é omitido, assume-se por **default** que se está usando o especificador de acesso *private*.

Os seguintes exemplos ilustram como estes especificadores são utilizados.

```
class B {
        int a;
protected:
        int b;
public:
        int c;
        void f();
};

void B::f() {
        a = 10;         // ok
        b = 10;         // ok
        c = 10;         // ok
}


B x;
int k = x.a;            // erro
k = x.b;                // erro
k = x.c;                // ok
```

```
x.f();                  // ok

class C: private B {
public:
        void g();
};

void C::g() {
        a = 10;         // erro
        b = 10;         // ok
        c = 10;         // ok
}

C y;
k = y.a;                // erro
k = y.b;                // erro
k = y.c;                // erro
y.f();                  // erro
y.g();                  // ok

class D: public B {
public:
        void h();
};

void D::h() {
        a = 10;         // erro
        b = 10;         // ok
        c = 10;         // ok
}

D w;
k = w.a;                // erro
k = w.b;                // erro
k = w.c;                // ok
w.f();                  // ok
w.h();                  // ok

class E: public C {
public:
        void i();
};



void E::i() {
        a = 10;         // erro
        b = 10;         // erro
        c = 10;         // erro
}
```

```
E z;
k = z.a;            // erro
k = z.b;            // erro
k = z.c;            // erro
z.f();              // erro
z.g();              // ok
z.i();              // ok

class F: public D {
public:
        void j();
};

void F::j() {
        a = 10;     // erro
        b = 10;     // ok
        c = 10;     // ok
}


F v;
k = v.a;            // erro
k = v.b;            // erro
k = v.c;            // ok
v.f();              // ok
v.h();              // ok
v.j();              // ok
```

5.6 Estruturas Internas a Classes

C++ possibilita que se criem estruturas e classes declaradas e visíveis apenas internamente a uma classe. A razão de se permitir este tipo de declaração é que pode ser necessário criar estruturas auxiliares para implementar outras classes. Normalmente, não faz sentido que estas estruturas sejam visíveis para outros clientes, uma vez que ela é específicamente relacionada com a classe para o qual foi criada. Por exemplo, na implementação de uma classe lista encadeada, é necessário criar uma estrutura nó da lista. Esta estrutura não deve ser pública, visto que ela está intimamente relacionado com o uso da classe lista.

```
class lista {
        struct no {
                no* prox;
                int info;
        }
        no* prim;
```

```
public:
        lista () { prim = 0 };
        void insere_inicio (int);
        void insere_fim (int);
        int remove ();
        int vazia ();
};
```

No exemplo acima a *struct no* só é visível internamente às funções membro da classe *lista*. Nenhuma outra função ou classe pode utilizar a estrutura nó.

A combinação deste mecanismo com a herança de classe através do especificador de acesso *private* abre novas possibilidades para a implementação de classes e permite diferenciar herança de propriedades do conceito de subtipos. Se uma classe é um subtipo de outra, todas as propriedades públicas da classe base devem ser públicas da classe derivada. Se apenas desejamos reutilizar código de uma classe sem que a nova classe seja considerada um subtipo, o uso do especificador de acesso *private* permite que definamos quais das propriedades de uma classe devem ser públicas. Por exemplo, a classe *pilha* não é um subtipo classe *lista*, pois nem todas operações de *lista* são operações de *pilha*. Contudo, podemos reusar operações de *lista* para implementar uma pilha.

```
class pilha: private lista {
public:
        pilha() {}
        void empilha (int valor) {
                lista::insere_inicio(int valor);
        }
        int pop() {
                return lista::remove_inicio();
        }
        lista::vazia;
};
```

5.7 Herança Múltipla

Herança Múltipla permite que uma classe derivada seja herdeira de duas ou mais classes bases imediatas, ou seja, filha de mais de uma classe. A sintaxe da declaração de uma classe é estendida de forma a permitir uma lista de classes bases e seus respectivos modificadores de acessabilidade. Por exemplo, considere a hierarquia da Figura 6:

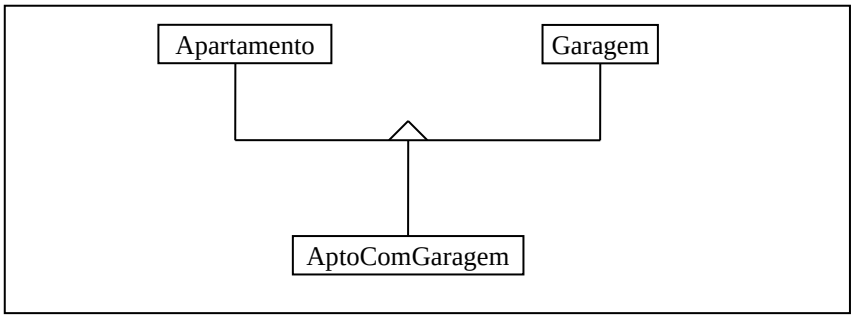


Figura 6 - Exemplo de Herança Múltipla

Uma possível implementação desta hierarquia em C++ é a seguinte:

```
class Apartamento {
protected:
        float area;
        unsigned qtde_quartos;
public:
        Apartamento(float, unsigned);
        void mostrar();
        friend void procura(unsigned, unsigned);
};

class Garagem {
protected:
        unsigned qtde_carros;
public:
        Garagem(unsigned);
        void mostrar();
        friend void procura(unsigned, unsigned);
};

class AptoComGaragem: Apartamento, Garagem {          // Herança Múltipla
public:
        AptoComGaragem(float, unsigned, unsigned);
        void mostrar();
};

/************** Funcoes da Classe Apartamento **************/

Apartamento::Apartamento(float metragem, unsigned dormitorios) {
        area = metragem;
        qtde_quartos = dormitorios;
}



void Apartamento::mostrar() {
        cout << "\n\n\t Numero de Dormitorios   : " << qtde_quartos
        << "\n\t Area do Apartamento     : "
        << setprecision(2) << area << "metros quadrados" ;
}

/************** Funcoes da Classe Garagem **************/
```

```
Garagem::Garagem(unsigned automoveis) {
        qtde_carros = automoveis;
}

void Garagem::mostrar() {
        cout << "Capacidade para  " << qtde_carros;
        if (qtde_carros > 1)
                cout << " Automoveis "
        else
                cout << "Automovel " ;
}

/************** Função da Classe AptoComGaragem **************/

AptoComGaragem::AptoComGaragem(float m2, unsigned td, unsigned ta)
        : Apartamento (m2,td), Garagem (ta) {}

void AptoComGaragem::mostrar() {
        Apartamento::mostrar();
        cout << "\n\t Garagem com " ;
        Garagem::mostrar();
}
```

Neste exemplo, a classe *AptoComGaragem* herda os membros das classes *Apartamento* e *Garagem*. Ao se criar uma instância da classe *AptoComGaragem* através da declaração

```
AptoComGaragem:: AptoComGaragem apt1202(130,4,2);
```

o construtor da classe derivada é chamado. Antes de executar seu código, esta função chama os construtores das classes bases *Apartamento* e *Garagem*, nesta ordem, em sua definição:

```
AptoComGaragem:: AptoComGaragem(float m2, unsigned td, unsigned ta)
        : Apartamento(m2, td), Garagem (ta) {}
```

5.7.1 Ambigüidades em Herança Múltipla

Herança Múltipla é conceitualmente direta, mas ela introduz algumas complexidades práticas nas linguagens de programação. Existem dois problemas relacionados a ambigüidade em Herança Múltipla: conflito entre nomes de classes bases diferentes e herança repetida. Conflitos ocorrem quando duas ou mais classes bases possuem membros homônimos. Em C++, tais conflitos devem ser resolvidos com uma qualificação explícita. Por exemplo:

```
class Surf {
        float notas[10];
public:
        virtual void ordena();
}
```

```
class Vela {
        float tempos[10];
public:
        virtual void ordena();
}

class WindSurf : public Surf, public Vela {
        char competicao;
};
```

Caso um objeto da classe *WindSurf* fosse criado e tentasse utilizar a função ordena() de uma classe base, haveria uma ambigüidade detectada pelo compilador.

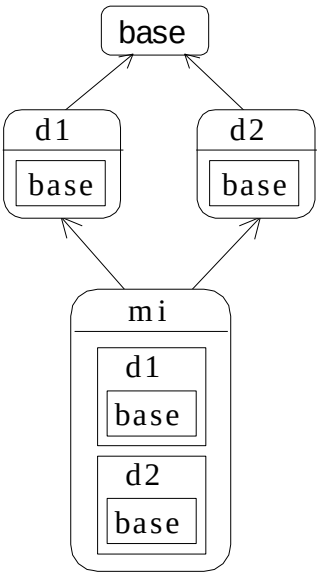
```
int main() {
        //...
        WindSurf corrida;
        corrida.ordena();       // erro de ambiguidade
        //...
}
```

Para evitar esse erro, deve-se especificar qual classe base está sendo referenciada pelo objeto da classe derivada. No exemplo, é o tipo de competição que identificará a classe base adequada. Sendo assim, pode-se eliminar a ambigüidade redefinindo-se a operação de ordenação na classe derivada *WindSurf* e utilizando-se o operador de resolução de escopo *::* para acessar as operações nas classes bases.

```
class WindSurf : public Surf, public Vela {
        char competicao;
public:
        virtual void ordena();
}

void WindSurf::ordena() {
        if (competicao == 'E' )
                Surf::ordena();
        else
                Vela::ordena();
}
```

Herança repetida ocorre quando uma classe é herdeira de duas ou mais classes que, por sua vez, são herdeiras de uma classe base comum. Neste caso os atributos da classe base comum serão herdados duplamente pela classe em questão. Este fato pode ser observado na figura a seguir:



Além de duplicar a memória utilizada para armazenar objetos da classe *mi*, a herança repetida pode provocar ambigüidade. Veja no exemplo seguinte:

```
//: C06:MultipleInheritance1.cpp
#include "../purge.h"
#include <iostream>
#include <vector>
using namespace std;

class MBase {
public:
        virtual char* vf() const = 0;
        virtual ~MBase() {}
};

class D1 : public MBase {
public:
        char* vf() const { return "D1"; }
};

class D2 : public MBase {
public:
        char* vf() const { return "D2"; }
};

class MI : public D1, public D2 {};  // erro devido a ambiguidade em vf

int main() {
        vector<MBase*> b;
        b.push_back(new D1);
        b.push_back(new D2);
        b.push_back(new mi);        // nao sabe qual Mbase herdado usar
        for(int i = 0; i < b.size(); i++)
                cout << b[i]->vf() << endl;
```

```
        purge(b);
} ///:~
```

Para contornar o primeiro erro, o programador deve disambigüar a função *vf()* através de sua redefinição na classe *mi*. C++ fornece um mecanismo especial, através da palavra *virtual* para resolver o segundo problema. Se uma classe é herdada precedida por um virtual somente um subobjeto daquela classe comporá o objeto da classe herdeira, independentemente se a classe é herdada múltiplas vezes ou não.

```
//: C06:MultipleInheritance2.cpp
// Virtual base classes
#include "../purge.h"
#include <iostream>
#include <vector>
using namespace std;

class MBase {
public:
        virtual char* vf() const = 0;
        virtual ~MBase() {}
};
class D1 : virtual public MBase {    // so herda MBase uma unica vez
public:
        char* vf() const { return "D1"; }
};
class D2 : virtual public MBase {   // so herda MBase uma unica vez
public:
        char* vf() const { return "D2"; }
};
// DEVE disambiguar vf() explicitamente
class MI : public D1, public D2 {
public:
        char* vf() const { return D1::vf();}
};
int main() {
        vector<MBase*> b;
        b.push_back(new D1);
        b.push_back(new D2);
        b.push_back(new MI); // OK
        for(int i = 0; i < b.size(); i++)
          cout << b[i]->vf() << endl;
        purge(b);
} ///:~
```

5.8 Funções e Classes Genéricas

Existem situações em programação onde é importante parametrizar uma determinada função ou classe pelo tipo do objeto que ela manipula. Por exemplo, um mesmo algoritmo de ordenação é aplicado independentemente do fato do tipo do elemento a ordenar ser inteiro, real, string ou mesmo um tipo composto. Da mesma

maneira, a estrutura e operações de uma classe lista independem do tipo dos elementos que compõem a lista.

Em muitas linguagens de programação, o programador é forçado a copiar (ou reescrever) o código destas funções e classes para cada um dos tipos desejados. Este procedimento costuma ser redundante, visto que o código é eminentemente o mesmo, e provocar erros, visto que durante a atualização do código copiado é necessário fazer pequenas modificações para ajustar a estrutura de dados e as operações ao novo tipo.

C++ fornece um mecanismo chamado *template* para tratar esses casos. Este mecanismo permite parametrizar uma função ou classe pelo tipo do elemento que ela manipula. Por exemplo, a função a seguir recebe dois parâmetros de um tipo não especificado e retorna o que tem maior valor.

```
#include <iostream.h>
template <class T>
        T max (T p, T s) {
        return p > s ? p : s;
}
```

Para utilizar esta função, basta chamá-la passando como parâmetros elementos de um mesmo tipo, visto que *p* e *s* são do tipo genérico *class T*. É importante notar que, para a função funcionar apropriadamente, é também necessário que o tipo do elemento que foi passado como parâmetro possua a comparação < como uma de suas operações, visto que a função vai necessitar comparar elementos deste tipo. O exemplo a seguir mostra como esta função pode ser utilizada.

```
main() {
        int c, a = 1, b = 2;
        char f, d = 'a', e = 'b';
        c = max (a, b);
        f = max (d, e);
        cout << " results: " << c << " - " << f << "\n";
}
```

O mesmo mecanismo pode ser usado para criar uma classe genérica. O exemplo a seguir mostra a implementação de uma classe genérica que define um tipo pilha.

```
template <class T, int tam>
class pilha {
   T vet[tam];
   int max;
   int top;
public:
   pilha(void);
   int vazia (void) { return top == -1; }
   T topo (void) { return vet[top]; }
   void empilha (T);
   void desempilha (void);
```

```
};
template <class T, int tam>
void pilha<T, tam>::pilha(void) {
      max = tam - 1;
      top = -1;
}
template <class T, int tam>
void pilha<T, tam>::empilha (T e){
  if (top == max) {
    cout << "pilha ja esta cheia\n";
  } else {
    vet[++top] = e;
  }
}
template <class T, int tam>
void pilha<T, tam>::desempilha (void){
  if (top == -1) {
    cout << "pilha ja esta vazia\n";
  } else {
     top--;
  }
}
```

Esta implementação do tipo pilha utiliza um vetor para armazenar os valores da pilha. Como pode ser observado, a classe não especifica qual o tipo do elemento da pilha. Isso só é definido quando da criação de um objeto desta classe. Este exemplo mostra ainda que a classe também é parametrizada por um inteiro que determina o tamanho máximo do vetor usado na pilha. Este valor só é definido no momento em que algum objeto da classe pilha é criado. O exemplo a seguir mostra o uso dessa classe na criação de uma pilha de *int* com tamanho máximo igual a três e uma pilha de *char* com tamanho máximo igual a 2:

```
#include "iostream.h"
#include "cgen.h"
void main () {
          pilha <int, 3> x;
          pilha <char, 2> y;
          x.empilha (1);
          x.empilha (2);
          x.empilha (3);
          x.empilha (4);
          x.desempilha( );
          cout << x.topo( ) << "\n" ;
          y.desempilha( );
          if (y.vazia( )) y.empilha('a');
          y.empilha('b');
          cout << y.topo( ) << "\n";
}
```

Note que, embora a pilha acima tenha sido usado com os tipos primitivos *int* e *char*, nada impede que ela seja também usada com um objeto de uma classe definida pelo programador. Outra observação interessante é que, ao contrário do que temos sempre visto, costuma-se colocar a especificação e a implementação de classes genéricas no arquivo cabeçalho (.h). Isso ocorre devido a forma de implementação dos templates que reutiliza código fonte e não código objeto. De fato, em alguns compiladores a separação de especificação e implementação de classes genéricas provoca erro de ligação.

5.9 Namespaces

Um dos problemas com a programação em C, é que quando programas atingem um certo tamanho fica cada vez mais difícil pensar em novos nomes para as entidades de computação globais. Pior ainda quando o programa é dividido em pedaços, cada qual é construído e mantido por um programador diferente. Como C só possui uma única região onde todos os identificadores globais são declarados, isto frequentemente provoca colisões de nomes.

C++ padrão inclui o mecanismo de namespaces para subdividir a região de identificadores globais em pedaços mais gerenciáveis. Cada conjunto de definições de C++ numa biblioteca ou programa é embutida numa *namespace*, e se alguma outra definição tem um identificador idêntico, mas numa outra *namespace*, então não existe colisão. A criação de uma *namespace* é muito similar a criação de uma classe:

```
//: C10:Header1.h
#ifndef HEADER1_H
#define HEADER1_H
namespace MyLib {
        extern int x;
        void f();
        // ...
}
#endif // HEADER1_H ///:~
```

Contudo, ao contrário de uma definição de classe, uma nova definição de uma *namespace* não implica numa redefinição da classe e sim numa continuação da definição anterior:

```
//: C10:Header2.h
#ifndef HEADER2_H
#define HEADER2_H
#include "Header1.h"
// Adiciona mais entidades a MyLib
namespace MyLib {               // Nao eh redefinicao!
        extern int y;
        void g();
        // ...
}
#endif // HEADER2_H ///:~
```

```
//: C10:Continuation.cpp
#include "Header2.h"
int main() {} ///:~
```

Uma *namespace* pode ser associada a um outro nome, de modo que o programador não tenha que usar nomes esquisitos ou grandes idealizados pelo criador da *namespace*.

```
//: C10:BobsSuperDuperLibrary.cpp
namespace BobsSuperDuperLibrary {
        class Widget { /* ... */ };
        class Poppit { /* ... */ };
        // ...
}
// Cria outro nome para a namespace pois e muito para teclar!!
namespace Bob = BobsSuperDuperLibrary;
int main() {} ///:~
```

O mecanismo de namespaces é conveniente e útil. Contudo, é necessário dar ciência da sua existência aos programas antes que eles as utilizem. Incluir apenas um arquivo cabeçalho provavelmente ocasionará mensagens de erros estranhas do compilador indicando que ele não é capaz de encontrar as declarações de ítens que você acabou de incluir com o arquivo cabeçalho. C++ fornece a palavra reservada *using* para indicar que se quer usar as declarações e definições de uma certa *namespace*.

```
using namespace std;
```

Toda a biblioteca padrão C++ está embutida na namespace *std*. Isto significa que sempre que a frase acima for incluída num programa, você pode usar qualquer entidade pertencente a biblioteca padrão incluída.

Existe uma correspondência entre a forma padronizada de inclusão de arquivos (sem o uso de .h), namespaces e a forma antiga (com o uso de .h):

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

é o mesmo que

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

5.10 Booleanos

Enquanto C não possui um tipo booleano, C++ padrão resgata este tipo de dado, chamado de *bool*. Antes que este tipo se tornasse parte de C++, muitos programadores tendiam a usar diferentes técnicas para produzir o efeito de booleanos. Isto produzia problemas de portabilidade e introduzia erros sutis.

O tipo *bool* pode assumir dois valores: *true* e *false*. O valor *true* converte para um e o valor *false* converte para zero. Adicionalmente, alguns mecanismos de C++ foram adaptados ao tipo booleano. Agora, enquanto os operadores lógicos e relacionais retornam booleanos, as expressões de comandos condicionais convertem implicitamente seus argumentos para valores booleanos.

5.11 Strings

O uso de vetores de char para tratar strings pode requerer grande esforço de programação para lidar com situações onde é necessário lidar com strings que possam aumentar de tamanho na medida da necessidade. A biblioteca padrão de C++ inclui uma classe *string* que foi projetada para cuidar (e esconder) todas as operações de gerenciamento de memória, cópia e concatenação de vetores de caracteres requeridas de um programador C. Essas manipulações são fonte constante de erros e desperdício de tempo.

```
//: C02:HelloStrings.cpp
// O basico da classe string de C++ padrao
#include <string>
#include <iostream>
using namespace std;
int main() {
        string s1, s2;                          // strings vazias
        string s3 = "Hello, World.";            // Inicializada
        string s4("I am");                      // Inicializada
        s2 = "Today";                           // Atribuicao
        s1 = s3 + " " + s4;                     // Concatenacao
        if (s1 >  s2)                           // Comparacao
                s1 += " 8 ";                    // Inclusao
        cout << s1 + s2 + "!" << endl;
} ///:~
```

O exemplo anterior mostra como strings podem ser criadas, inicializadas, atribuídas, concatenadas e comparadas de uma forma direta, sem que o programador necessite saber como estas operações são implementadas.

5.12 Streams

Não seria interessante você poder tratar todos os tipos de dispositivos de armazenamento de dados (entrada e saída padrão, arquivos e blocos de memória) de uma maneira equivalente de modo a só manipulá-los com uma única interface de operações? Esta é a idéia por detrás de *iostream*s. Elas são muito mais fáceis, seguras e frequentemente eficientes do que as funções avulsas da biblioteca padrão *stdio* de C.

A biblioteca padrão *iostream* é geralmente a primeira que novos programadores C++ aprendem a usar. Isto ocorre porque este pacote define automaticamente objetos chamados *cin* e *cout*, que aceitam dados que devem ser lidos ou escritos na entrada e saída padrão. Para enviar dados para a saída padrão usa-se o

operador sobrecarregado <<. Para receber dados da entrada padrão usa-se o operador sobrecarregado >>.

```
int i;                          cout << "i = ";
cin >> i;                       cout << i;
float f;                        cout << "\n";
cin >> f;                       cout << "f = ";
char c;                         cout << f;
cin >> c;                       cout << "\n";
char buf[100];                  cout << "c = ";
cin >> buf;                     cout << c;
                                cout << "\n";
                                cout << "buf = ";
                                cout << buf;
                                cout << "\n";
```

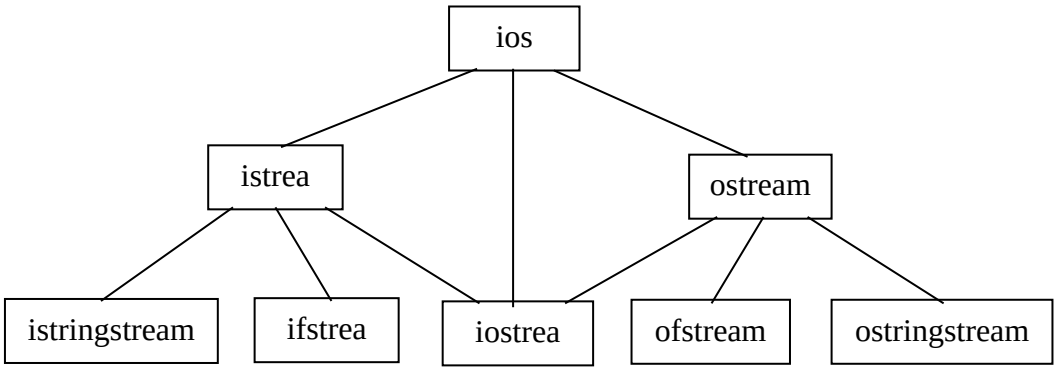
que é o mesmo que

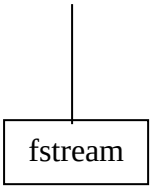
```
cin >> i >> f >> c >> buf;
cout << "i = " <<  i << "\nf = "<< f << "\nc = "<<
c << "\nbuf = " << buf << "\n";
```

É importante ter em mente que o uso de *cin* apresenta as mesmas características da entrada com *scanf* em C, isto é, o uso do delimitador espaço para separar entradas e a possibilidade de ler strings além do tamanho reservado para elas. Para ler strings que contenham brancos com segurança C++ fornece o método *getline*, que permite ler uma linha inteira

```
cin.getline(buf, sz);   // terceiro parâmetro é o delimitador - default = \n
```

C++ oferece, na biblioteca padrão iostream, uma hierarquia de classes para manipulação do sistema de entrada e saída através de streams. A figura seguinte apresenta parte desta hierarquia. Uma stream em C++ é um objeto que formata e armazena bytes e que corresponde a uma classe desta hierarquia. A classe raiz desta hierarquia é *ios*, a qual contém dados e operações para todas as classes derivadas. As classes *istream* e *ostream* fornecem respectivamente operações básicas de entrada e saída e são usadas como classes bases para as demais classes da hierarquia. As classes *ifstream* e *ofstream* são usadas para manipular arquivos. As classes *istringstream* e *ostringstream* para manipular strings. A classe *fstream* permite criar e manter arquivos que requeiram acesso de leitura e escrita.





Para abrir arquivos para leitura e escrita em C++, é necessário incluir a biblioteca padrão <*fstream*>. Para abrir um arquivo para leitura, é necessário criar um objeto da classe *ifstream*. Para abrir um arquivo para escrita, é necessário criar um objeto da classe *ofstream*. Uma vez que você tenha aberto o arquivo, pode-se ler ou escrever nele como se estivesse manipulando qualquer objeto da classe *iostream*.

```
//: C02:Scopy.cpp
// Copia um arquivo sobre outro, uma linha de cada vez
#include <string>
#include <fstream>
using namespace std;
int main() {
        ifstream in("Scopy.cpp");      // Abrir para leitura
        ofstream out("Scopy2.cpp"); // Abrir para escrita
        string s;
        while(getline(in, s)) // Discarta o caracter de nova linha
        out << s << "\n"; // ... adiciona de novo o caracter de nova linha
} ///:~
```

Outro exemplo interessante copia um arquivo completo em uma string:

```
//: C02:FillString.cpp
#include <string>
#include <iostream>
#include <fstream>
using namespace std;
int main() {
        ifstream in("FillString.cpp");
        string s, line;
        while(getline(in, line))
        s += line + "\n";
        cout << s;
} ///:~
```

Por causa da natureza dinâmica das strings, não é necessário se preocupar com a alocação de memória para a string.

5.13 Tratamento de Exceções

Uma exceção é um evento que ocorre durante a execução de um programa que interrompe o fluxo normal das instruções. Muitos tipos de erros podem causar exceções, tais como falha de hardware e tentativa de acesso a posições fora dos

limites de um vetor. Mas exceções não estão relacionadas exclusivamente com erros. Condições excepcionais, tal como a chegada ao fim de um arquivo também são exceções.

O tratamento de exceções é bastante direto em situações onde uma determinada condição é verificada e já se sabe exatamente o que se deve fazer porque toda a informação necessária está disponível naquele contexto. Você simplesmente as trata no ponto onde elas ocorrem. Esta postura tende a carregar o código, comprometendo a legibilidade do programa.

Em situações onde não existe informação disponível no contexto onde a condição foi verificada, é necessário passar a informação local sobre a exceção para um contexto mais externo onde esta informação exista. Uma maneira de fazer isto é retornando códigos que representam as exceções. Contudo, uma das razões pelas quais é prática comum ignorar condições de exceção em programação é relacionado com o fato de ser extremamente tedioso e dispersante escrever código que faça checagem extensiva de erros. A proliferação de código para tratamento de erro, além de ser extremamente cansativa, ainda dificulta em muito a legibilidade do programa.

Uma das principais características de C++ é o mecanismo de tratamento de exceções, que objetiva tornar esta tarefa menos tediosa e deixar o código do programa mais simples e claro. Com o mecanismo de tratamento de exceções de C++:

1. O código de tratamento de exceções não fica misturado com o código que cumpre as funcionalidades do programa. Você escreve o código que você quer que execute e mais tarde, numa seção separada, você escreve o código para lidar com os problemas. Se você faz múltiplas chamadas a uma função você trata os erros daquela função um única vez, em um único local.
2. Exceções não podem ser ignoradas. Se uma função necessita enviar uma mensagem de erro para quem chamou aquela função, ele envia um objeto representando aquele erro para fora da função. Se quem chamou a função não trata o erro, o objeto é enviado para o próximo nível de escopo até que alguma função trate o erro.

5.13.1 Sinalizando uma Exceção

Em C++, se você encontra uma situação excepcional no seu código, você pode enviar informação sobre o contexto corrente para um contexto mais externo através da criação de um objeto contendo a informação e do seu envio para fora de seu contexto corrente. Isto se chama sinalizar uma exceção. O exemplo a seguir mostra um exemplo de como isso é feito:

```
throw meuErro ("algo de errado aconteceu");
```

O objeto meuErro é criado especialmente para a sinalização de exceções. É possível utilizar qualquer tipo (inclusive os pré-definidos) para sinalizar exceções. O comando indicado pela palavra reservada *throw* primeiramente constrói um objeto que normalmente não existiria na execução normal do programa. Este objeto é posteriormente retornado pela função onde o comando foi escrito. Ao contrário do

retorno de funções, onde o fluxo de execução continua a partir do ponto onde a função foi chamada, uma sinalização de exceção pode desviar o fluxo para um ponto que pode estar bem distante de onde a função foi chamada. Além disso, apenas objetos que foram criados até o ponto onde a exceção é sinalizada serão destruídos (ao contrário de retorno de funções, que assume que todos os objetos no escopo devem ser destruídos). Obviamente, o objeto criado pela exceção também será destruído no ponto apropriado. Por fim, uma função pode sinalizar várias exceções e criar objetos de tipos diferentes para cada uma delas tanto quanto for necessário. Em regra geral, você vai sinalizar um objeto de tipo diferente para cada tipo diferente de exceção.

Se você está dentro de uma função e uma exceção é sinalizada (ou uma função que foi chamada sinaliza uma exceção), esta função será encerrada durante o processo de sinalização. Se você não quer que isso ocorra, você pode definir um bloco especial dentro da função onde você tenta realizar uma funcionalidade (e gerar exceções potenciais). Para criar este bloco, deve-se usar a palavra reservada *try*:

```
try {
  // Code that may generate exceptions
}
```

Se tivéssemos de checar os erros sem usar tratadores de exceção, cada chamada de função deveria ser envolvida com código de teste, mesmo se a mesma função fosse invocada várias vezes. Com o *try*, todas as chamadas de função são colocadas dentro de um bloco sem que seja necessária alguma checagem de erro. Isto quer dizer que seu código é mais fácil de escrever e ler porque o objetivo do seu código não se confunde com a checagem de erros.

5.13.1 Capturando e Tratando uma Exceção

Obviamente, a exceção sinalizada deve ser capturada e tratada em algum lugar. Este lugar é chamado de tratador de exceção. Existe um tratador para cada tipo de exceção que se pode capturar. Tratadores de exceção seguem imediatamente o bloco *try* e são denotados pela palavra reservada *catch*:

```
try {
    // codigo que pode sinalizar excecoes
} catch (tipo1 id1) {
    // trata excecoes de tipo1
    // ...
    throw;          // repropaga a excecao
} catch (tipo2 id2) {
    // trata excecoes de tipo2
} catch ( ... )
    // trata excecoes de qualquer tipo - deve ser o ultimo
}
```

Cada cláusula *catch* é como uma pequena função que recebe um único argumento (não pode ser mais que um) de um determinado tipo. O identificador do

parâmetro pode ser omitido se o objeto não for necessário para tratar a exceção (isto é, o tipo da exceção já fornece informação suficiente para tratar a exceção).

Se uma exceção é sinalizada, o primeiro tratador cujo tipo de parâmetro case com o tipo da exceção será executado e a exceção será tratada (a busca pelo tratador é interrompida uma vez que a execução de um tratador é finalizada). Portanto, somente a a primeira cláusula que casa que é executada. Note que, dentro de um bloco *try*, várias chamadas de função podem gerar a mesma exceção, porém somente um tratador é necessário.

Algumas vezes é necessário criar um tratador de exceções que capture qualquer tipo de exceção. Isto é realizado usando uma elipse como parâmetro do *catch*. Como este tratador capturará qualquer exceção, quando for usado, ele deve ser colocado ao final da lista de tratadores para evitar que algum tratador nunca seja chamado.

Outras vezes, pode ser necessário resinalizar a exceção que acabou de ser capturada. Isto é feito através do uso de *throw* sem nenhum objeto associado. Quando isto ocorre, qualquer outro tratador do bloco *try* é também ignorado e a busca pelo tratador da exceção passa para o contexto mais externo seguinte. É importante saber que o objeto da exceção é preservado intacto, de modo que o tratador do nível mais externo também é capaz de extrair informação deste objeto.

Se nenhum dos tratadores de exceção de um determinado bloco *try* casa com a exceção, a busca pelo tratador passa para o contexto mais externo seguinte, isto é, a função ou bloco *try* que envolve o bloco *try* que falhou na captura da exceção. É importante ter atenção nestes casos porque a localização do novo bloco *try* para busca não é sempre óbvia. Este processo continua até que, em algum nível, um tratador case com a exceção. Neste ponto, a exceção é tratada e não é feita nenhuma busca adicional. Se nenhum tratador é encontrado em qualquer nível, o programa é abortado. Isto também ocorre se uma nova exceção é sinalizada antes de que uma exceção previamente sinalizada atinja o seu tratador (a causa mais comum para isto ocorre quando o construtor do objeto da exceção causa ele mesmo uma exceção).

O exemplo a seguir mostra como o mecanismo de tratamento de exceções pode ser utilizado:

```
class Array {
    // ...
public:
    const int max = 3200;
    class Range {
    public:
        int index;
       Range (int i): index (i) { }
    }
    class Size { }
    Array (int sz);
    int& operator[] (int i);
    // ...
}
```

```
Array::Array (int sz) {
      // ...
      if (sz < 0 || max > sz) throw Size( );
      // ...
}

int& Array::operator[] (int i) {
     // ...
      if (i < 0 || i > sz) throw Range( i );
      // ...
}

void f ( ) {
     // ...
     try {
          usa_Arrays ( );   // qualquer funcao que use vetores
     }
     catch (Array::Size) {
         // ... ajusta
         f ( );
     }
     catch (Array::Range r) {
         cerr << "indice fora dos limites do vetor: " << r.index << '\n';
         exit (99);
     }
     // chega aqui se nao houver excecoes ou ao fim da excecao Size
     // ...
}

void f1 ( ) {
     try {
         f2( );
     }
     catch (Array::Size) {
         // ...
     }
}

void f2 ( ) {
     try {
         usa_Arrays ( );       // qualquer funcao que use vetores
     }
     catch (Array::Range) {
         // ...
     }
}
```

Um outro exemplo usando herança e amarração tardia:

```
class MathErr {
    // ...
    virtual void debugPrint ( );
}
class Overflow: public MathErr { // ... };
class Underflow: public MathErr { // ... };
class ZeroDivide: public MathErr { // ... };

void g ( ) {
    try {
        //...
    }
    catch (Overflow) {
        // ... trata ocorrencias de Overflow e subclasses
    }
    catch (MathErr m) {
        // ... qualquer MathErr que nao e overflow
        m.debugPrint ( );
    }
    //...
}
```

5.14 Coleções e Iteradores da Biblioteca Padrão C++

Esta seção objetiva introduzir as classes de coleções e iteradores que fazem parte da biblioteca padrão C++. É importante esclarecer que existe uma pequena confusão a respeito destas classes, que são frequentemente referidas como parte da STL (Standard Template Library). STL foi o nome dado por Alex Stepanov para apresentar sua biblioteca para o Comitê de Padronização de C++, em 1994. O nome foi amplamente aceito pela comunidade de programadores C++. O Comitê de padronização de C++ decidiu integrá-la à biblioteca padrão de C++, porém fazendo um número significativo de modificações. O desenvolvimento da STL continuou na Silicon Graphics (SGI), de onde surgiu a SGI STL. Esta biblioteca diverge sutilmente da biblioteca padrão em muitos pontos. Portanto, embora seja uma confusão popular, a biblioteca padrão C++ não inclui a STL. Aqui não discutiremos em detalhes as classes destas bibliotecas. Vamos apresentar uma introdução que será válida tanto para uma quanto para outra.

Existem situações onde não se sabe quantos objetos serão necessários para resolver um determinado problema ou quanto tempo estes objetos irão durar. Nestes casos, você não saberá como armazená-los. A solução para estes problemas é criar um novo tipo de objeto que se expandirá sempre que for necessário acomodar qualquer coisa dentro dele. Este tipo de objeto é chamado uma coleção. Portanto, coleções são classes que descrevem objetos capazes de armazenar outros objetos. Assim, quando uma destas situações aparece basta criar um objeto do tipo coleção e deixá-lo cuidar dos detalhes.

Boas LPs orientadas a objetos fornecem um conjunto de coleções como parte da sua biblioteca padrão. Em algumas bibliotecas, uma coleção genérica é considerada

boa para todas as necessidades. Em outras (como C++) a biblioteca fornece diferentes tipos de coleções para diferentes tipos de uso: um vetor para acesso consistente a todos os elementos; uma lista encadeada para a inserção consistente de elementos em qualquer ponto da lista. Estas bibliotecas podem incluir coleções que modelam conjuntos, filas, pilhas, árvores, tabelas hash, etc.

Todas coleções têm alguma maneira efetiva e diferenciada de colocar, armazenar, recuperar e operar sobre os elementos dentro dela. Normalmente, o modo como se colocam coisas numa coleção é através de uma função de inserção ("push"ou "add" ou um nome similar). Já o modo de recuperar elementos de uma coleção é mais variado. Por exemplo, se a coleção é uma entidade tal como um vetor, o programador deve poder usar indexação.

Para se ter uma idéia do que são coleções vamos examinar três das mais básicas coleções da biblioteca padrão C++. Um *vector* é uma sequência linear que permite o acesso aleatório rápido aos seus elementos. Contudo, é custoso inserir um elemento no meio da sequência e é também custoso alocar memória adicional. Um *deque* também é uma sequência linear que permite acesso aleatório quase tão rápido quanto *vector*, mas é significativamente mais rápida quando necessita alocar nova memória e permite incluir facilmente novos elementos no início e fim da coleção (*vector* só permite inclusão ao final). Uma *list* é uma sequência linear onde é muito custoso acessar elementos aleatoriamente e pouco custoso inserir um elemento no meio da sequência. Como todas estas coleções apresentam uma funcionalidade básica similar, na maioria das situações bastará escolher uma delas e somente experimentar com as outras quando se estiver buscando eficiência.

Para acessar elementos de um *vector* ou *deque* é possível usar indexação, porém o mesmo não pode ser feito com uma *list*. Em algumas situações, um único modo de acesso pode ser muito restritivo. Além disso, seria interessante possuir uma interface comum de acesso entre as diversas coleções.

Um iterador é um objeto de uma classe que abstrai o processo de percorrer os elementos de uma sequência e apresentá-los ao usuário do iterador. Ele permite que você selecione cada elemento da sequência sem ter de saber os detalhes de estrutura interna de armazenamento da coleção, seja ela um *vector*, uma *stack*, uma *list*, ou qualquer outra. Através do iterador, a coleção é simplesmente abstraída a uma sequência. Tal característica é poderosa porque nos permite aprender uma única interface que funciona com todas as coleções e parcialmente porque permite as coleções serem usadas intercambiadamente. Em outras palavras, ela fornece flexibilidade para mudar facilmente a coleção sem pertubar o código do programa usuário.

```
//: C04:Stlshape.cpp
#include <vector>
#include <iostream>
using namespace std;

class Shape {
public:
  virtual void draw() = 0;
  virtual ~Shape() {};
```

```
};

class Circle : public Shape {
public:
   void draw() { cout << "Circle::draw\n"; }
   ~Circle() { cout << "~Circle\n"; }
};

class Triangle : public Shape {
public:
   void draw() { cout << "Triangle::draw\n"; }
   ~Triangle() { cout << "~Triangle\n"; }
};

class Square : public Shape {
public:
   void draw() { cout << "Square::draw\n"; }
   ~Square() { cout << "~Square\n"; }
};

typedef std::vector<Shape*> Container;
typedef Container::iterator Iter;          // iterador definido dentro de vector

int main() {
   Container shapes;                       // cria vector de shapes
   shapes.push_back(new Circle);           // coloca elemento ao final da coleção
   shapes.push_back(new Square);
   shapes.push_back(new Triangle);
   for(Iter i = shapes.begin(); i != shapes.end(); i++)
      (*i)->draw();          // poderia usar i->draw() --- * e -> sao sobrecarregados
   // ...
   for(Iter j = shapes.begin();  j != shapes.end();  j++)
     delete *j;               // colecoes nao chamam os destrutores dos objetos
 } ///:~
```

É bastante interessante notar que é possível mudar o tipo da coleção trocando apenas duas linhas do programa acima. Ao invés de incluir <*vector*>, poderia-se incluir <*list*> e substituir o primeiro *typedef* por:

```
typedef std::list<Shape*> Container;
```

Isto é possível não por causa de uma interface estabelecida por herança (pode parecer surpreendente, mas não existe herança entre as coleções), mas por causa de uma interface comum estabelecida através de convenções adotadas pela biblioteca padrão, de modo que essa mudança possa ser feita.

Existem duas razões para que se possa escolher coleções. Primeiramente, coleções fornecem diferentes tipos de interfaces e comportamento externo. Uma *stack* tem uma diferente interface e comportamento que uma *queue*, que é diferente de um *set* ou *list*. Uma destas coleções pode se ajustar melhor ao seu problema do que outra.

Em segundo lugar, diferentes coleções possuem diferentes eficiências para certas operações. Por exemplo, inserção de elementos no meio de uma sequência tem diferenças radicais no custo se usamos um *vector* ou uma *list*. Como os iteradores fornecem uma interface comum para as coleções, você pode implementar um protótipo usando uma determinada coleção e depois modificar para uma coleção mais eficiente quando da implementação final do programa, com impacto mínimo no seu código.

Bibliografia

[1] Stroustrup, B.; 1988; "What is Object-Oriented Programming?"; IEEE Software vol 5(3); p.10

[2] Meyer, B.; 1988; "Object-Oriented Software Construction"; New York, NY: Prentice Hall.

[3] Stroustrup, B.; 1991; "The C++ Programming Language"; Second Edition; Addison Wesley.

[4] Booch, G.; 1994; "Object-Oriented Analysis and Design with Applications"; Second Edition ; The Benjamin/Cummings Publishing Company.

[5] Pohl, I.; 1989; "C++ for Programmers"; The Benjamin/Cummings Publishing Company".

[6] Montenegro, F. & Pacheco, R.; 1994; "Orientação a Objetos em C++"; Ciência Moderna.

[7] Gorlen, K. & Orlow, S. & Plexico, P.; 1990; "Data Abstraction and Object-Oriented Programming in C++"; Addison Wesley.

[8] Rumbaugh, J. & Premerlani, W. & Eddy, F. & Lorensen, W.; 1991; Object-Oriented Modeling and Design"; Englewood Cliffs, New Jersey: Prentice Hall.

[9] Eckel, B.; 1999; "Thinking in C++", Volume I, 2nd Edition.

[10] Eckel, B; 2000; "Thinking in C++", Volume II, 1st Edition.