Programação Orientada por Objetos – Tópico 1

***Temática**: Introdução a programação orientada por objetos*

Actividade 1: Familiarização com o conceito de POO e qualidade de software, implementação do 1º programa em C++

Competências a desenvolver:

- Compreender a importância da qualidade de software
- Aprender o conceito do paradigma da programação orientada por objetos
- Aprender a instalar e configurar o ambiente Eclipse para trabalhar com C++
- Criar o primeiro programa básico em C++

Qualidade do software

O principal objetivo da Engenharia de Software é contribuir para a produção de programas de qualidade. Esta qualidade, porém, não uma ideia simples; mas sim como um conjunto de noções e fatores. É desejável que os programas produzidos sejam rápidos, confiáveis, modulares, estruturados, modularizados, etc. Esses qualificadores descrevem dois tipos de qualidade:

De um lado, considera-se aspectos como eficiência, facilidade de uso, extensibilidade, etc. Estes elementos podem ser detectados pelos utilizadores do sistema. A esses fatores atribui-se a qualidade externa do software.

Por outro lado, existe um conjunto de fatores do programa que só profissionais de computação podem detectar. Por exemplo, legibilidade, modularidade, etc. A esses fatores atribui-se a qualidade interna do sofware.

Na realidade, qualidade externa do programa em questão é que importa quando de sua utilização. No entanto, os elementos de qualidade interna são a chave para a conquista da qualidade externa.

Fatores de qualidade externa

- **Corretude**: É a condição do programa de produzir respostas adequadas e corretas cumprindo rigorosamente suas tarefas de acordo com sua especificação.
- **Robustez**: É a capacidade do programa de funcionar mesmo sob condições anormais.
- **Extensibilidade**: É definida como a facilidade com que o programa pode ser adaptado para mudanças em sua especificação.
    - Neste aspecto, dois princípios são essenciais: Simplicidade de *design* e descentralização (quanto mais autónomos são os módulos de uma arquitetura, maior será a probabilidade de que uma alteração implicará na manutenção de um ou poucos módulos).
- **Capacidade de reuso**: É a capacidade do programa de ser reutilizado, totalmente ou em partes, para novas aplicações.
- **Compatibilidade**: É a facilidade com que o programa pode ser combinado com outros.

Outros aspectos

Os aspectos resumidos acima são aqueles que podem ser beneficiados com a boa utilização das técnicas de orientação por objetos. No entanto, existem outros aspectos que não podem ser esquecidos:

- **Eficiência**: É o bom aproveitamento dos recursos computacionais como processadores, memória, dispositivos de comunicação, etc. Apesar de não explicitamente citado anteriormente, este é um requisito essencial para qualquer produto. A linguagem C++ em particular, por ser tão eficiente quanto C, permite o desenvolvimento de sistemas eficientes.
- **Portabilidade**: É a facilidade com que um programa pode ser transferido de uma plataforma (hardware, sistema operacional, etc.). A nível de linguagem, C++ satisfaz este fator por ser uma linguagem padronizada com implementações nas mais diversas plataformas.
- **Facilidade de uso**: É a facilidade de aprendizagem de como o programa funciona, sua operação, etc.

Modularidade - qualidade interna

Alguns fatores apresentados na seção anterior podem ser beneficiados com o uso de orientação por objetos. São eles: corretude, robustez, extensibilidade, reuso e compatibilidade. A medida que se olha para estes cinco aspectos, dois subgrupos surgem:

- Extensibilidade, reuso e compatibilidade demandam arquiteturas que sejam flexíveis, design descentralizado e a construção de módulos coerentes conectados por interfaces bem definidas.
- Corretude e robustez são favorecidos por técnicas que suportam o desenvolvimento de sistemas baseados em uma especificação precisa de requisitos e limitações.

Da necessidade da construção de arquiteturas de programas que sejam flexíveis, surge a questão de tornar os programas mais modulares.

Critérios para a modularidade

Decomposição: O método deve ajudar a reduzir a aparente complexidade de problema inicial pela sua decomposição em um conjunto de problemas menores conectados por uma estrutura simples. Uma exemplificação deste tipo de design é o chamado método top-down.

Composição: O método que satisfaz o critério de composição favorece a produção de elementos de programas que podem ser livremente combinados com os outros de forma a produzir novos sistemas (reutilização), possivelmente em um ambiente bem diferente daquele em que cada um destes elementos foi criado.

Entendimento: Um método que satisfaz o critério de entendimento ajuda a produção de módulos que podem ser separadamente compreendidos pelos desenvolvedores. Este critério é especialmente relevante quando se tem em vista o aspecto da manutenção.

Continuidade: Um método de design satisfaz a continuidade se uma pequena mudança na especificação do problema resulta em alterações em um único ou poucos módulos.

Proteção: Um método de *design* satisfaz a proteção se este provê a arquitetura de isolamento quando da ocorrência de condições anómalas em tempo de execução. Ao aparecimento de situações anormais, seus efeitos ficam restritos àquele módulo ou pelo menos se propagar a poucos módulos vizinhos.

Princípios de modularidade

Estabelecidos os critérios de modularidade, alguns princípios surgem e devem ser observados cuidadosamente para se obter modularidade:

- **Linguística modular**: Este princípio expressa que o formalismo utilizado para expressar o *design*, programas, etc. deve suportar uma visão modular; isto é, módulos devem corresponder às unidades sintáticas da linguagem utilizada. Onde a linguagem utilizada pode ser qualquer linguagem de programação, de design de sistemas, de especificação, etc.

Este princípio segue diversos critérios mencionados anteriormente:

- **Decomposição**: Se pretende-se dividir o desenvolvimento do sistema em tarefas separadas, cada uma destas deve resultar em uma unidade sintática bem delimitada. (compiláveis separadamente no caso de linguagens de programação)
- **Composição**: Só pode-se combinar coisas como unidades bem delimitadas.
- **Proteção**: Somente pode haver controle do efeito de erros se os módulos estão bem delimitados.
- **Poucas interfaces**: Cada módulo deve se comunicar o mínimo possível com outros.
- **Pequenas interfaces**: Este princípio relaciona o tamanho das interfaces e não suas quantidades. Isto quer dizer que se dois módulos possuem canal de comunicação, estes devem trocar o mínimo de informação possível; isto é, os canais da comunicação intermodular devem ser limitados.
- **Interfaces explícitas**: Além de impor limitações no número de módulos que se comunicam e na quantidade de informações trocadas, há a imposição de que se explicite claramente esta comunicação.

Este princípio segue de diversos critérios mencionados anteriormente:

- **Decomposição e composição**: Se um elemento é formado pela composição ou decomposição de outros, as conexões devem ser bem claras entre eles.
- **Entendimento**: Como entender o funcionamento de um módulo A se seu comportamento é influenciado por outro módulo B de maneira não clara?
- **Ocultação de informação**: A aplicação deste princípio assume que todo módulo é conhecido através de uma descrição oficial e explícita; ou melhor sua interface. A ideia geral é simples: a interface deve ser uma descrição do funcionamento do módulo. Tudo que for relacionado com sua implementação não deve ser público.

Segundo a norma ISO/IEC 9126-1 (figura 1), para qualquer requisito de qualidade interna ou externa deve ser possível especificá-lo utilizando-se apenas as seis características. A programação orientada por objetos, como veremos ao longo desta UC, faculta uma série de mecanismos que ajudam a ir de encontro com os principais fatores de qualidade necessários no desenvolvimento de software.

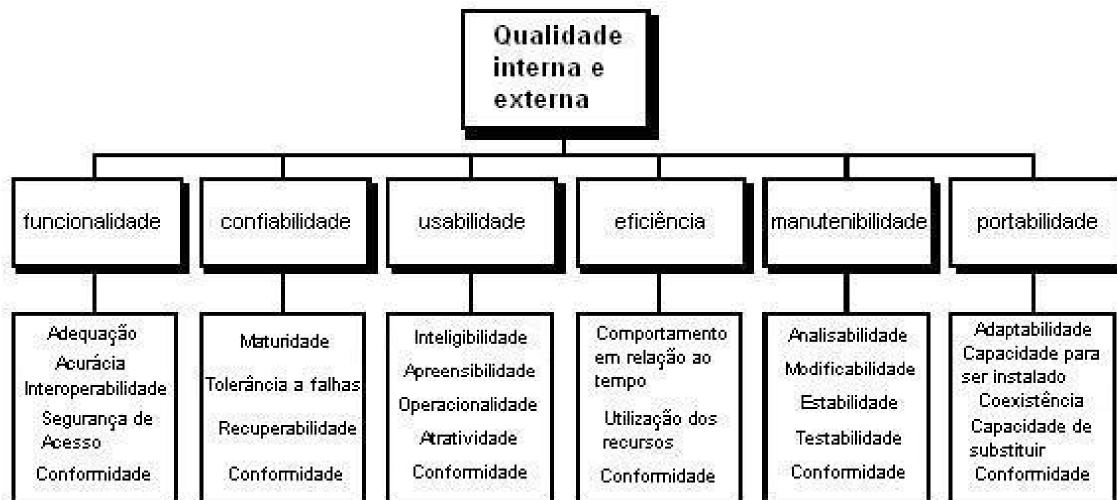

Figura 1. ISO/IEC 9126-1 - Modelo de Qualidade Interna e Externa

Tipos abstratos de dados

O conceito de tipo foi um passo importante no sentido de atingir uma linguagem capaz de suportar programação estruturada. Porém, até agora, não é possível usar uma metodologia na qual os programas sejam desenvolvidos por meio de decomposições de problemas baseadas no reconhecimento de abstrações. Para a abstração de dados adequada a este propósito não basta meramente classificar os objetos quanto a suas estruturas de representação; em vez disso, os objetos devem ser classificados de acordo com o seu comportamento esperado. Esse comportamento é expresso em termos das operações que fazem sentido sobre esses dados; estas operações são o único meio de criar, modificar e se ter acesso aos objetos.

Classes em C++

Uma classe em C++ é o elemento básico sobre o qual toda orientação por objetos está apoiada. Em primeira instância, uma classe é uma extensão de uma estrutura, que passa a ter não apenas dados, mas também funções. A idéia é que tipos abstratos de dados não são definidos pela sua representação interna, e sim pelas operações sobre o tipo. Então não há nada mais natural do que incorporar estas operações no próprio tipo.Estas operações só fazem sentido quando associadas às suas representações.

O paradigma da orientação a objetos

Objeto: é uma abstração encapsulada que tem um estado interno dado por uma lista de atributos cujos valores são únicos para o objeto. O objeto também conhece uma lista de mensagens que ele pode responder e sabe como responder cada uma. Encapsulamento e abstração são definidos mais à frente.

Mensagem: é representada por um identificador que implica em uma ação a ser tomada pelo objeto que a recebe. Mensagens podem ser simples ou podem incluir parâmetros que afetam

como o objeto vai responder à mensagem. A resposta também é influenciada pelo estado interno do objeto.

Classe: é um modelo para a criação de um objeto. Inclui em sua descrição um nome para o tipo de objeto, uma lista de atributos (e seus tipos) e uma lista de mensagens com os métodos correspondentes que o objeto desta classe sabe responder.

Instância: é um objeto que tem suas propriedades definidas na descrição da classe. As propriedades que são únicas para as instâncias são os valores dos atributos.

Método: é uma lista de instruções que define como um objeto responde a uma mensagem em particular. Um método tipicamente consiste de expressões que enviam mais mensagens para objetos. Toda mensagem em uma classe tem um método correspondente.

Traduzindo estas definições para o C++, classes são estruturas, objetos são variáveis do tipo de alguma classe (instância de alguma classe), métodos são funções de classes e enviar uma mensagem para um objeto é chamar um método de um objeto.

Resumindo:

Objetos são instâncias de classes que respondem a mensagens de acordo com os métodos e atributos, descritos na classe.

Instalação do ambiente para desenvolvimento em C++

Existem vários IDEs para desenvolvimento em C++. Nesta UC, é sugerida a utilização da plataforma Eclipse, que por ser multi-plataforma, é possível correr em todos os sistemas operativos. Para baixar e instalar uma versão, aceda o seguinte link: http://www.eclipse.org/downloads/

O Eclipse possui várias versões. O ideal será baixar e instalar a versão "Eclipse IDE for C/C++ Developers". Se não baixar essa, será necessário instalar o CDT para tornar o IDE passível para C++. Neste caso, após ter efetuado a instalação, vá ao menu "Ajuda" e aceda a opção "Instalação de software…". A figura 2 ilustra a janela que se abrirá. Indique o caminho onde deve ser encontrado o CDT de acordo com a versão do Eclipse que tenha instalado (Work with). O caminho é dado por: http://download.eclipse.org/tools/cdt/releases/versão_do_eclipse_instalado, como por exemplo, http://download.eclipse.org/tools/cdt/releases/indigo ou http://download.eclipse.org/tools/cdt/releases/juno.

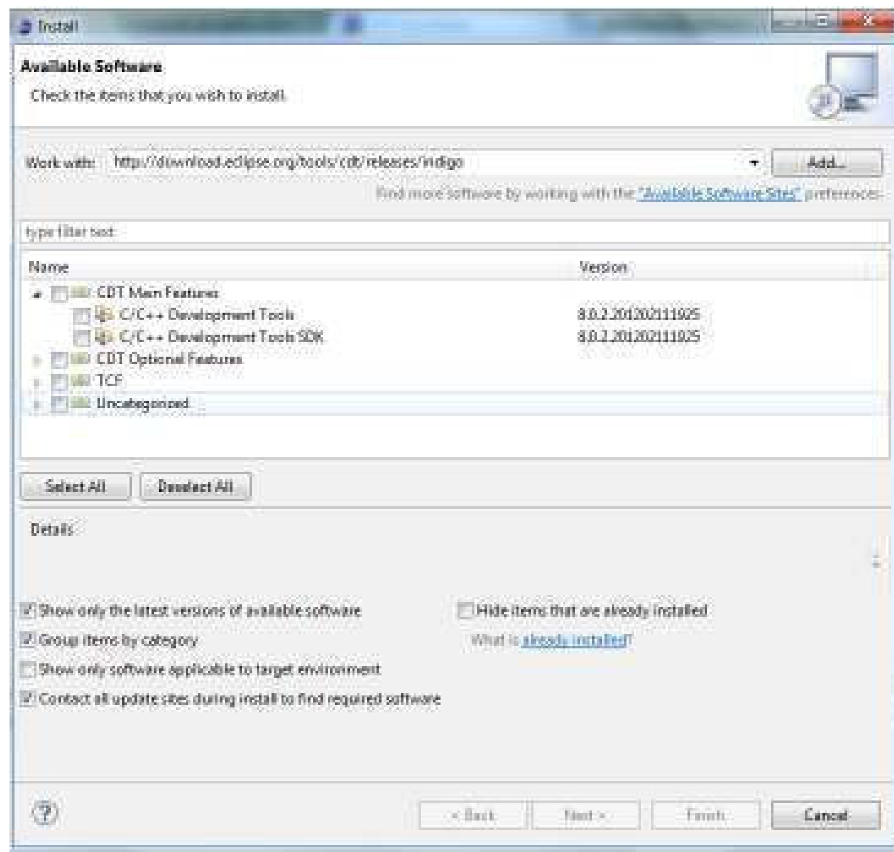

Figura 2. Janela de instalação de software do Eclipse

Vá nas opções e garanta que são instaladas as componentes marcadas a vermelho na figura 3.

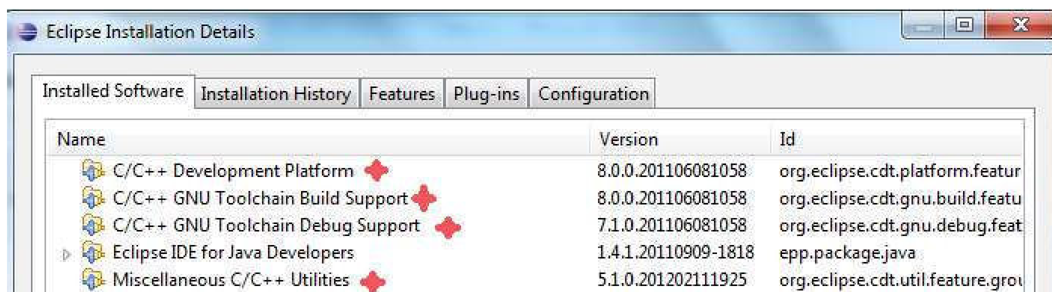

Figura 3. Pacotes a serem instalados

No caso do seu sistema operativo ser Windows, ainda será necessário instalar o compilador de C++. Neste caso, baixe e instale o MinGW e MSYS que estão disponibilizados na plataforma. Execute o instalador de MinGW com a opção “current” activada e garanta que pelo menos as duas primeiras opções estão escolhidas (veja a figura 4 e 5).

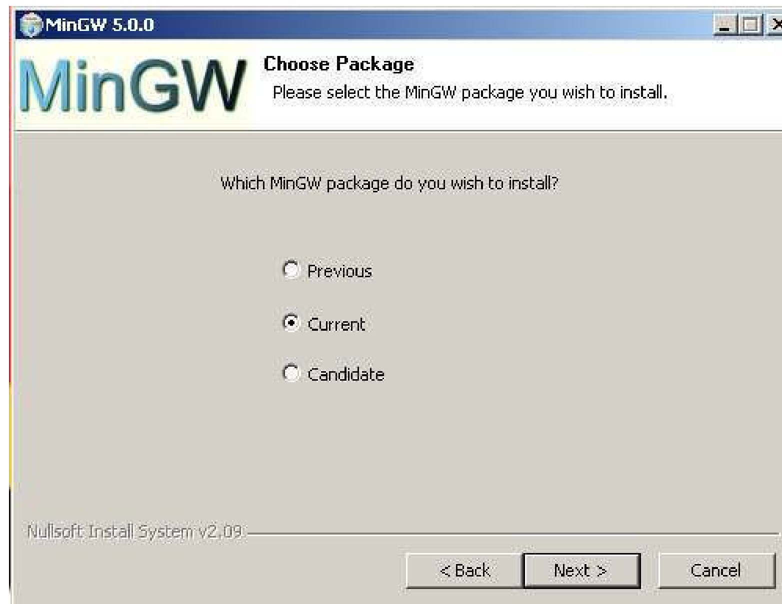

Figura 4. MinGW instalação parte inicial

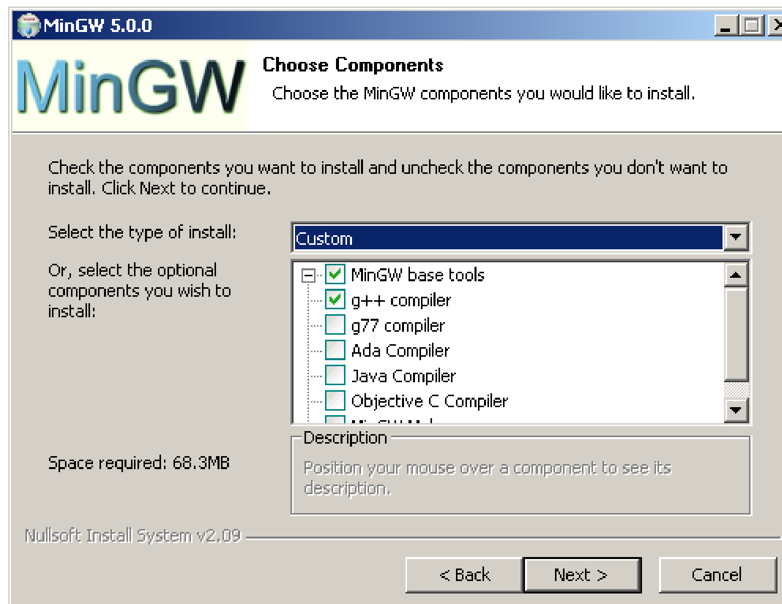

Figura4. MinGW escolha de componentes

Não altere os parâmetros de instalação e quando acabar, execute o instalador do MSYS. Neste caso, será aberta uma janela em MS-DOS com perguntas as quais deverá sempre confirmar, aceitando. A última pergunta é sobre a localização da instalação do MinGW. Deve indicar o caminho correto (p. ex. C:\MinGW), tendo cuidado de evitar confusões como o uso de letras maiúsculas/minúsculas ou ainda escrever de forma apropriada para o MS-DOS, nomes de caminhos que tenham mais que 8 letras ou espaços em branco (p. ex. C:/Program Files/MinGW passa a ser C:/PROGRA~1/mingw).

Por fim, deverá ainda adicionar a variável de sistema "path" da sua máquina, os caminhos onde a pasta "bin" está localizada para cada uma das instalações, isto é, C:\MinGW\bin e C:\MSYS\1.0\bin, no caso de a instalação ter sido feita na raiz de C. Reinicie a máquina e teste se o MinGW e MSYS ficou bem instalado, abrindo uma janela MS-DOS e executando os seguintes

comandos: gcc --version e make --version. Se tudo estiver bem, serão listadas as versões instaladas.

Primeiro programa em C++ - ALÔ MUNDO!

Uma tradição seguida em muitos cursos de programação é iniciar o aprendizado de uma nova linguagem como programa "Alo, Mundo!". Trata-se de um programa elementar, que simplesmente exibe esta mensagem na console. Mesmo sendo um programa muito simples, AloMundo.cpp serve para ilustrar diversos aspectos importantes da linguagem. AloMundo.cpp ilustra também a estrutura básica de um programa C++. No caso do Eclipse, você pode cria-lo de forma automática, pois este projeto está incluído. Vá ao menu File>>New>>C++ Project e opte pelo projeto "Hello World". Mande construí-lo (compila-lo, construindo o executável) e depois execute-o, acendendo os menus Project>>Build All e Run. O resultado é a mensagem aparecer na console.

Vamos tentar compreende-lo:

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

int main() {
	cout << "!!!Hello World!!!" << endl;
	return 0;
}
```

#include <iostream>

O símbolo # é uma chamada de atenção ao compilador a dizer que aquela linha é para o preprocessador, depois temos o "*include*" (que basicamente diz para incluir código). Os ficheiros a serem incluídos contêm as declarações das funções e definições que o código fonte utilizará. Neste caso, é o *iostream*. (in+out+stream, "fluxo de entrada e saída", padrão). Na maioria das vezes, os arquivos de cabeçalho fazem parte de uma biblioteca, que podem ser as já disponíveis com o compilador (identificadas como *standard*), criadas por nós ou compradas.

O ficheiro/arquivo **iostream** está envolvido em < >, isto significa que o preprocessador deve procurar o ficheiro/arquivo no sítio/diretório usual (que é onde o compilador usa como padrão para os "includes"). Se tivéssemos o ficheiro/arquivo iostream envolvido em "" significaria que o preprocessador deveria procurá-lo dentro de uma lista de diretórios de inclusão, "includes", iniciando pelo diretório atual.

using namespace std;

namespaces são espaços de nomes dentro do código, que funcionam, entre outras coisas, como um meio de evitar duplicação de nomes dentro de um projeto extenso, que geralmente contam com inúmeros arquivos.

O C++ usa os *namespaces* para organizar os diferentes nomes usados nos programas. Cada nome usado no ficheiro/arquivo biblioteca "**standard iostream**" faz parte do "namespace" chamado de std.

O objeto de saída padrão, cout, está definido dentro do "namespace" std, ele é um objeto da classe "ostream" "output stream", para acessá-lo temos que referenciá-lo como "std::cout". Para evitar

que tenhamos que informar "std::" todas as vezes que precisarmos usar os recursos deste "namespace", podemos informar que estamos usando-o dentro do arquivo atual, conforme vemos na linha declarada no início deste tópico.

O "namespace" permite que as variáveis sejam localizadas em certas regiões do código. Declarar o "namespace std" permite que todos os objectos e funções da biblioteca "standard input-output" possam ser usados sem qualquer qualificações específicas, desta maneira, não é mais necessário o uso de "std::".

int main(){}

Como na linguagem C, a função principal de entrada do programa a partir do sistema operacional é a função *main*. Por isso mesmo ela é obrigatória em qualquer programa. Se não existisse uma "main function", não haveria entrada para que o sistema iniciasse o programa.

Todas as funções são declaradas e usadas com o operador ( ). As funções declaradas como membros de uma classe de objetos podem ser chamadas de métodos.

O *int* significa que a função vai retornar um inteiro.

cout << "Hello world! << endl;

(c+out) Podemos utilizar este objeto porque pusemos o *header* e o *namespace std*. As informações serão direcionadas através do iostream, um subsistema de entrada e saída da biblioteca padrão. O que este objeto nos permite é enviar o que temos entre aspas para a saída (out), que é o monitor neste caso.

O **cout** envia dados para o "standard output device", que é usualmente o monitor, a abstração do elemento de saída padrão é observada na presença de um objeto que representa a saída física de dados para o meio externo. **endl** indica o fim de linha e dá um salto para a seguinte.

return 0

Faz com que a função retorne o valor zero, como esta função é a principal do programa, por onde o sistema operativo/operacional iniciou a execução do mesmo, este retorno é recebido pelo sistema, é comum que valores diferentes de zero sejam interpretados como erro do programa.

Programação Orientada por Objetos – Tópico 2

***Temática**: Descrição de recursos de OO não relacionados as classes*

***Actividade 2**: Conhecer aspectos da programação OO que são transversais a programação procedimental*

Competências a desenvolver:

- Aprender os recursos disponíveis em C++ que não estão directamente relacionados com a programação OO, mas herdados em grande parte da linguagem C
- Aprender a implementar um programa em C++ com a utilização de funções e todos os demais recursos não directamente relacionados com a POO

Descrição de recursos de C++ não relacionados às classes

TIPOS DE DADOS

A linguagem C++ tem cinco tipos básicos de dados, que são especificados pelas palavras-chave:

```
char
int
float
double
bool
```

O tipo *char* (caractere) é usado para texto. O tipo *int* é usado para valores inteiros. Os tipos *float* e *double* expressam valores de ponto flutuante (fracionários). O tipo *bool* expressa os valores verdadeiro (true) e falso (false).

É importante ressaltar que embora C++ disponha do tipo *bool*, qualquer valor diferente de zero é interpretado como sendo verdadeiro (true). O valor zero é interpretado como sendo falso (false).

O exemplo abaixo cria variáveis dos tipos básicos e exibe seus valores.

```
// Tipos.cpp
// Ilustra os tipos básicos de C++.
#include <iostream.h>
int main()
{
// Declara e inicializa uma variável char.
       char cVar = 't';
// Declara e inicializa uma variável int.
       int iVar = 298;
// Declara e inicializa uma variável float.
       float fVar = 49.95;
// Declara e inicializa uma variável double.
       double dVar = 99.9999;
// Declara e inicializa uma variável bool.
       bool bVar = (2 > 3); // False.
// O mesmo que: bool bVar = false;
// Exibe valores.
```

```
        cout << "cVar = " << cVar << "\n";
        cout << "iVar = " << iVar << "\n";
        cout << "fVar = " << fVar << "\n";
        cout << "dVar = " << dVar << "\n";
        cout << "bVar = " << bVar << "\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:
cVar = t
iVar = 298
fVar = 49.95
dVar = 99.9999
bVar = 0

O tipo CHAR

O tipo *char* (caractere) geralmente tem o tamanho de um byte, o que é suficiente para conter 256 valores. Observe que um *char* pode ser interpretado de duas maneiras:

- Como um número pequeno (0 a 255)
- Como um elemento de um conjunto de caracteres, como ASCII.

Exemplo

```
// TChar.cpp
// Ilustra o uso do tipo char.
#include <iostream.h>
int main()
{
// Exibe o alfabeto minúsculo.
      for(char ch = 97; ch <= 122; ch++)
              cout << ch << " ";
      return 0;
} // Fim de main()
```

Saída gerada por este programa:

a b c d e f g h i j k l m n o p q r s t u v w x y z

SEQUÊNCIAS DE ESCAPE

Alguns caracteres podem ser representados por combinações especiais de outros caracteres. Essas combinações são conhecidas como sequências de escape, porque "escapam" do significado normal do caracter. A lista abaixo representa algumas sequências de escape mais comuns:

\n caracter de nova linha
\t caracter de tabulação (tab)
\b caracter backspace
\" aspa dupla
\' aspa simples
\? ponto de interrogação
\\ barra invertida

Exemplo

```
// Escape.cpp
// Ilustra o uso de sequências de escape.
#include <iostream.h>
int main()
{
// Exibe frases usando sequências de escape.
       cout << "\"Frase entre aspas\"\n";
       cout << "Alguma duvida\?\n";
       return 0;
} // Fim de main()
```

Saída gerada por este programa:

"Frase entre aspas"
Alguma duvida?

VARIÁVEIS UNSIGNED

Em C++, os tipos inteiros existem em duas variedades: *signed* (com sinal) e *unsigned* (sem sinal). A ideia é que, às vezes é necessário poder trabalhar com valores negativos e positivos; outras vezes, os valores são somente positivos. Os tipos inteiros (*short*, *int* e *long*), quando não são precedidos pela palavra *unsigned* sempre podem assumir valores negativos ou positivos. Os valores *unsigned* são sempre positivos ou iguais a zero.

Como o mesmo número de bytes é utilizado para os inteiros *signed* e *unsigned*, o maior número que pode ser armazenado em um inteiro *unsigned* é o dobro do maior número positivo que pode ser armazenado em um inteiro *signed*.

A tabela abaixo ilustra os valores de uma implementação típica de C++:

Tipo	Tamanho (em bytes)	Valores
unsigned short int	2	0 a 65.535
short int	2	-32.768 a 32.767
unsigned long int	4	0 a 4.294.967.295
long int	4	-2.147.483.648 a 2.147.483.647
int (16 bits)	2	-32.768 a 32.767
int (32 bits)	4	-2.147.483.648 a 2.147.483.647
unsigned int (16 bits)	2	0 a 65.535
unsigned int (32 bits)	4	0 a 4.294.967.295
char	1	256 valores de caracteres
float	4	1,2e-38 a 3,4e38
double	8	2,2e-308 a 1,8e308

Exemplo

```
// TamUns.cpp
```

```
// Ilustra o tamanho // das variáveis unsigned.
#include <iostream.h>
int main()
{
        cout << "*** Tamanhos das variaveis ***\n";
        cout << "Tamanho de unsigned int = " << sizeof(unsigned int)
        << " bytes.\n";
        cout << "Tamanho de unsigned short int =
        " << sizeof(unsigned short) << " bytes.\n";
        cout << "Tamanho de unsigned char = " << sizeof(unsigned char)
        << " bytes.\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

*** Tamanhos das variaveis ***
Tamanho de unsigned int = 4 bytes.
Tamanho de unsigned short int = 2 bytes.
Tamanho de unsigned char = 1 bytes.

O TIPO STRING

Uma das atividades mais comuns em qualquer tipo de programa é a manipulação de strings de texto. Por isso, a biblioteca padrão C++ oferece um tipo, chamado string, que permite realizar diversas operações úteis com strings de texto.

O exemplo abaixo é uma reescrita do programa elementar AloMundo.cpp, usando o tipo string.

```
// AloStr.cpp
// Ilustra o uso do tipo string.
#include <iostream.h>
int main()
{
// Declara e inicializa uma variável do tipo string.
        string aloTar = "Alo, Tarcisio!";
        cout << aloTar;
} // Fim de main()
```

Saída gerada por este programa:

Alo, Tarcisio!

CONCATENANDO STRINGS

O tipo *string* permite o uso do operador + para concatenar (somar) strings. O exemplo abaixo mostra como isso é feito.

Exemplo

```
// SomaStr.cpp
// Ilustra o uso do operador + com o tipo string.
#include <iostream.h>
int main()
{
// Declara e inicializa algumas variáveis do tipo string.
```

```
        string s1 = "Agua mole ";
        string s2 = "em pedra dura ";
        string s3 = "tanto bate ";
        string s4 = "ate que fura";
// Exibe usando // o operador +
        cout << s1 + s2 + s3 + s4 + "!!!\n\n";
} // Fim de main()
```

Saída gerada por este programa:

Agua mole em pedra dura tanto bate ate que fura!!!

COMENTÁRIOS

O primeiro recurso apresentado é simplesmente uma forma alternativa de comentar o código. Em C++, os caracteres // iniciam um comentário que termina no fim da linha na qual estão estes caracteres.

DECLARAÇÃO DE VARIÁVEIS

Em C++ as variáveis podem ser declaradas em qualquer ponto do código. O objetivo deste recurso é minimizar declarações de variáveis não inicializadas. Se a variável pode ser declarada em qualquer ponto, ela pode ser sempre inicializada na própria declaração.

DECLARAÇÃO DE TIPOS

Em C++, as declarações abaixo são equivalentes:

```
struct a {
        // ...
};

typedef struct a {
        // ...
} a;
```

A simples declaração de uma estrutura já permite que se use o nome sem a necessidade da palavra reservada *struct* , como mostrado abaixo:

```
struct a {};
void f(void)
{
        a a2; // em C++ a palavra struct
}
```

As estruturas são variáveis que funcionam como registos (são compostos por outras variáveis que funcionam como “campos”).

DECLARAÇÃO DE UNIÕES

As uniões são variáveis que podem assumir diferentes tipos e partilham o mesmo endereço e espaço de memória. Em C++ as uniões podem ser anónimas. Uma união anónima é uma declaração da forma:

```
union { lista dos campos };
```

Neste caso, os campos da união são usados como se fossem declarados no âmbito da própria união. Um uso mais comum para uniões anónimas é dentro de estruturas (*struct*).

```
struct A {
        int tipo;
        union {
           int inteiro;
           float real;
           void *ponteiro;
        };
};
```

Os três campos da união podem ser acedidos diretamente, da mesma maneira que o campo tipo.

PROTÓTIPOS DE FUNÇÕES

Em C++ uma função só pode ser usada se esta já foi declarada. Para usar uma função que não tenha sido definida antes da chamada é necessário usar protótipos. Os protótipos de C++ incluem não só o tipo de retorno da função, mas também os tipos dos parâmetros:

```
void f(int a, float b);// protótipo da função f
void main(void)
{
        f(1, 4.5); // o protótipo possibilita a utilização de f aqui
}
void f(int a, float b)
{
        printf("%.2f\n", b+a+0.5);
}
```

As funções representam um dos blocos construtivos da linguagem C++. Outro bloco construtivo básico de C++ são as classes de objetos. Todo programa C++ tem obrigatoriamente pelo menos uma função, chamada main(). Todo comando executável em C++ aparece dentro de alguma função. Dito de forma simples, uma função é um grupo de comandos que executa uma tarefa específica, e muitas vezes retorna (envia) um valor para o comando que a chamou. A definição de uma função consiste de um cabeçalho e de um corpo. O cabeçalho contém o tipo retornado, o nome da função e os parâmetros que ela recebe. Os parâmetros de uma função permitem que passemos valores para a função.

Exemplo

```
// FunSimp.cpp
// Ilustra o uso de uma função simples.
#include <iostream.h>
int Soma(int i, int j)
{
        cout << "Estamos na funcao Soma().\n";
        cout << "Valores recebidos: \n";
        cout << "i = "<< i << ", j = "<< j << "\n";
        return (i + j);
} // Fim de Soma(int, int)
int main()
{
        cout << "Estamos em main()\n";
        int x, y, z;
```

```
        cout << "\nDigite o primeiro num. + <Enter>";
        cin >> x;
        cout << "\nDigite o segundo num. + <Enter>";
        cin >> y;
        cout << "Chamando funcao Soma()...\n";
        z = Soma(x, y);
        cout << "Voltamos a main()\n";
        cout << "Novo valor de z = " << z << "\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Estamos em main()
Digite o primeiro num. + <Enter>48
Digite o segundo num. + <Enter>94
Chamando funcao Soma()...
Estamos na funcao Soma().

Valores recebidos:

i = 48, j = 94
Voltamos a main()
Novo valor de z = 142

Muitas das funções que usamos em nossos programas já existem como parte da biblioteca padrão que acompanha o compilador C++. Para usar uma dessas funções, é necessário incluir no programa o arquivo que contém o protótipo da função desejada, usando a diretiva #*include*. Para as funções que nós mesmos escrevemos, precisamos escrever o protótipo.

FUNÇÕES: VARIÁVEIS LOCAIS

Além de podermos passar variáveis para uma função, na forma de argumentos, podemos também declarar variáveis dentro do corpo da função. Essas variáveis são chamadas locais, porque somente existem localmente, dentro da função. Quando a função retorna, a variável deixa de existir.

As variáveis locais são definidas da mesma forma que as outras variáveis. Os parâmetros da função são também considerados variáveis locais, e podem ser usados exatamente como se tivessem sido definidos dentro do corpo da função.

Exemplo

```
// Local.cpp
// Ilustra o uso de variáveis locais e função
#include <iostream.h>
// Protótipo.
// Converte temperatura em graus
// Fahrenheit para graus centígrados.
double FahrParaCent(double);
int main()
{
        double tempFahr, tempCent;
        cout << "\n*** Conversao de graus Fahrenheit "
```

```
"para graus Centigrados ***\n";
       cout << "Digite a temperatura em Fahrenheit: ";
       cin >> tempFahr;
       tempCent = FahrParaCent(tempFahr);
       cout << "\n" << tempFahr << " graus Fahrenheit = "
<< tempCent << " graus Centigrados.\n";
       return 0;
} // Fim de main()
// Definição da função.
double FahrParaCent(double fahr)
{
// Variável local.
       double cent;
       cent = ((fahr - 32) * 5) / 9;
       return cent;
} // Fim de FahrParaCent(double fahr)
```

Saída gerada por este programa:

*** Conversao de graus Fahrenheit para graus Centigrados ***
Digite a temperatura em Fahrenheit: 65
65 graus Fahrenheit = 18.3333 graus Centigrados.

VARIÁVEL DENTRO DE UM BLOCO

Dissemos que uma variável local declarada dentro de uma função tem escopo local. Isso quer dizer que essa variável é visível e pode ser usada somente dentro da função na qual foi declarada.

C++ permite também definir variáveis dentro de qualquer bloco de código, delimitado por chaves { e }. Uma variável declarada dentro de um bloco de código fica disponível somente dentro desse bloco.

FUNÇÕES: PARÂMETROS COMO VARIÁVEIS LOCAIS

Os parâmetros de uma função são variáveis locais à função. Isso significa que alterações feitas nos argumentos recebidos não afetam os valores originais desses argumentos. Isso é conhecido como passagem por valor. O que acontece é que a função cria cópias locais dos argumentos recebidos, e opera sobre essas cópias. Tais cópias locais são tratadas exatamente como as outras variáveis locais.

Qualquer expressão válida em C++ pode ser usada como argumento, inclusive constantes, expressões matemáticas e lógicas e outras funções que retornem um valor do tipo correto. Os parâmetros de uma função não precisam ser todos do mesmo tipo.

Exemplo

```
// Param.cpp
// Ilustra os parâmetros de uma função como variáveis locais.
#include <iostream.h>
// Protótipo.
void troca(int, int);
int main()
{
// Declara variáveis locais em main()
```

```
        int var1 = 10, var2 = 20;
        cout << "Estamos em main(), antes de troca()\n";
        cout << "var1 = " << var1 << "\n";
        cout << "var2 = " << var2 << "\n";
// Chama a função.
        troca(var1, var2);
        cout << "Estamos em main(), depois de troca()\n";
        cout << "var1 = " << var1 << "\n";
        cout << "var2 = " << var2 << "\n";
        return 0;
} // Fim de main()
// Definição da função.
void troca(int var1, int var2)
{
// Exibe os valores.
        cout << "Estamos em troca(), antes da troca\n";
        cout << "var1 = " << var1 << "\n";
        cout << "var2 = " << var2 << "\n";
// Efetua a troca.
        int temp;
        temp = var1;
        var1 = var2;
        var2 = temp;
// Exibe os valores.
        cout << "Estamos em troca(), depois da troca\n";
        cout << "var1 = " << var1 << "\n";
        cout << "var2 = " << var2 << "\n";
} // Fim de troca(int, int)
```

Saída gerada por este programa:

Estamos em main(), antes de troca()
var1 = 10
var2 = 20
Estamos em troca(), antes da troca
var1 = 10
var2 = 20
Estamos em troca(), depois da troca
var1 = 20
var2 = 10
Estamos em main(), depois de troca()
var1 = 10
var2 = 20

RETORNANDO VALORES

Em C++, todas as funções, ou retornam um valor, ou retornam *void*. A palavra-chave *void* é uma indicação de que nenhum valor será retornado. Para retornar um valor de uma função, escrevemos a palavra-chave *return*, seguida do valor a ser retornado. O valor pode ser também uma expressão que retorne um valor. Eis alguns exemplos válidos de uso de *return*:

return 10;
return (a > b);
return(funcaoArea(larg, comp));

A execução do programa volta imediatamente para a função chamadora, e quaisquer comandos que venham depois da palavra-chave *return* não serão executados. Podemos ter diversos comandos *return* dentro de uma função. Porém, cada vez que a função for chamada, somente um dos *return* será executado.

UMA FUNÇÃO MEMBRO DE STRING

Vimos que a biblioteca padrão de C++ contém o tipo *string*, usado na manipulação de *strings* de texto. Na verdade, esse tipo é implementado como uma classe de objetos, um conceito fundamental em C++. Embora ainda não tenhamos estudado os objetos em C++, podemos usá-los de forma mais ou menos intuitiva, para ter uma ideia do poder e da praticidade que representam.

Por exemplo, os fluxos de entrada e saída *cin* e *cout*, são objetos de C++. O tipo *string* também é um objeto. Objetos contêm operações que facilitam sua manipulação. Essas operações são similares a funções que ficam contidas no objeto, por isso são chamadas de **funções membro**. Para chamar uma função membro de um objeto, usamos o operador ponto. Por exemplo, a linha abaixo chama a função membro substr() de um objeto da classe *string*, para aceder uma *substring* contida nesta *string*.

sobreNome = nome.substr(9, 5);

Outra operação comum com *strings* é substituir parte de uma *string*. Como se trata de uma operação com *strings*, nada mais lógico que esta operação esteja contida nos objetos da classe *string*. Esta operação é feita com uma função membro de *string* chamada replace(). O exemplo abaixo mostra como ela pode ser utilizada.

Exemplo

```
// ReplStr.cpp
// Ilustra outras funções de strings.
#include <iostream.h>
int main()
{
// Declara e inicializa uma variável do tipo string.
       string nome = "Tarcisio Lopes";
// Exibe.
       cout << "Meu nome = " << nome << "\n";
// Utiliza a função membro replace() para
// substituir parte da string.
// A parte substituída começa em 0 e tem o comprimento 8
       nome.replace(0, 8, "Mateus");
// Exibe nova string.
       cout << "Nome do meu filho = " << nome << "\n";
} // Fim de main()
```

Saída gerada por este programa:

Meu nome = Tarcisio Lopes
Nome do meu filho = Mateus Lopes

FUNÇÕES INLINE

Funções *inline* são comuns em C++, tanto em funções globais como em métodos. Estas funções têm como objetivo tornar mais eficiente, em relação à velocidade, o código que chama estas funções. Elas são tratadas pelo compilador quase como uma macro: a chamada da função é substituída pelo corpo da função. Este tipo de otimização normalmente só é utilizado em funções pequenas e o compilador pode ou não aceitar que a função seja tratada como tal (gera um aviso nesse caso).

```
inlinedouble quadrado(double x)
{
        return x * x;
}
```

Assim como as funções, os métodos também podem ser declaradas como *inline*. No caso de métodos não é necessário usar a palavra reservada *inline*. A regra é a seguinte: funções com o corpo declarado dentro da própria classe são tratadas como *inline*; funções declaradas na classe mas definidas fora não são *inline*.

Um módulo C++ é composto por dois arquivos, o arquivo com as declarações exportadas (.h) e outro com as implementações (.c, .cc ou .cpp). Tipicamente, uma função exportada por um módulo tem o seu protótipo no .h e sua implementação no .c. Isso porque um módulo não precisa conhecer a implementação de uma função para usá-la. Mas isso só é verdade para funções não *inline*. Por causa disso, funções *inline* exportadas precisam ser implementadas no .h e não mais no .c.

VALORES DEFAULT PARA PARÂMETROS DE FUNÇÕES

Em C++ existe a possibilidade de definir valores *default* para parâmetros de uma função. Quando uma função usa valores *default*, os parâmetros com *default* devem ser os últimos e a declaração do valor *default* só pode aparecer uma vez. Por exemplo:

```
void impr( char* str, int x = -1, int y = -1)
```

Esta definição indica que a função *impr* tem três parâmetros, sendo que os dois últimos têm valores *default*. A função acima pode ser utilizada das seguintes maneiras:

```
impr( "especificando a posição", 10, 10 ); // x=10, y=10
impr( "só x", 20 ); // x=20, y=-1
impr( "nem x nem y" ); // x=-1, y=-1
```

SOBRECARGA DE NOMES DE FUNÇÕES

O nome de função pode ter mais de um significado, dependendo do contexto. Ter mais de um significado quer dizer que um nome de função pode estar associado a várias implementações diferentes. Com sobrecarga isto é possível; o compilador escolhe que implementação usar dependendo do contexto. No caso de funções o contexto é o tipo dos parâmetros (só os parâmetros, não os resultados):

```
display( "string" );
display( 123 );
display( 3.14159 );
```

Cada chamada, invocaria respectivamente uma implementação diferente:

```
void display( char *v ) { printf("%s", v); }
```

```
void display( int v ) { printf("%d", v); }
void display( float v ) { printf("%f", v); }
```

ALOCAÇÃO DE MEMÓRIA

Para a gestão da memória, existem os operadores, *new* e *delete*, que são duas palavras reservadas. O operador *new* aloca memória e o *delete* desaloca.

```
int * i2 = new int;
```

A alocação de um vetor também é bem simples:

```
int * i4 = new int[10];
```

A liberação da memória alocada é feita pelo operador *delete*. Este operador pode ser usado em duas formas; uma para desalocar um objeto e outra para desalocar um vetor de objetos:

```
delete i2; // alocado com new
delete [] i4; // alocado com new[]
```

OPERADOR DE ÂMBITO

C++ possui um operador que permite o acesso a nomes declarados em âmbitos que não sejam o corrente. O operador de âmbito possibilita o uso de nomes que não estão no âmbito corrente, o que pode ser usado neste caso:

```
char *a;
void main(void)
{
        int a;
        a = 23; // a é a variável local
        ::a = "abc"; // a é a variável global
}
```

A sintaxe deste operador é a seguinte:

âmbito::nome,

onde *âmbito* é o nome da classe onde está declarado no nome *nome*. Se âmbito não for especificado, como no exemplo acima, o nome é procurado no espaço global.

Outros usos deste operador são apresentados nas seções sobre classes aninhadas, campos de estruturas *static* e métodos *static*.

USANDO TYPEDEF

Às vezes, o processo de declaração de variáveis pode se tornar tedioso, repetitivo e sujeito a erros. Isso acontece, por exemplo, se usamos muitas variáveis do tipo *unsigned short int* em um programa. C++ permite criar um novo nome para esse tipo, com o uso da palavra-chave *typedef*.

Eis a forma de uso de *typedef*: **typedef unsigned short int USHORT**;

A partir daí, podemos usar USHORT, ao invés de *unsigned short int.*

CONSTANTES COM #DEFINE

Muitas vezes, é conveniente criar um nome para um valor constante. Este nome é chamado de constante simbólica. A forma mais tradicional de definir constantes simbólicas é usando a diretiva de preprocessador #define:

#define PI 3.1416

Observe que neste caso, PI não é declarado como sendo de nenhum tipo em particular (float, double ou qualquer outro). A diretiva #define faz simplesmente uma substituição de texto. Todas as vezes que o preprocessador encontra a palavra PI, ele a substitui pelo texto 3.1416.

Exemplo

```
// DefTst.cpp
// Ilustra o uso de #define e typedef
#include <iostream.h>
// Para mudar a precisão, basta alterar #define.
#define PI 3.1416
//#define PI 3.141593
// Cria um sinônimo usando typedef.
typedef unsigned short int USHORT;
int main()
{
        USHORT raio = 5;
        cout << "Area do circulo " << "de raio 5 = " << PI * raio *
raio << "\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Area do circulo de raio 5 = 78.54

CONSTANTES COM CONST

Embora a diretiva #define funcione, C++ oferece uma forma melhor de definir constantes simbólicas: usando a palavra-chave const.

const float PI = 3.1416;

Este exemplo também declara uma constante simbólica chamada PI, mas desta vez o tipo de PI é declarado como sendo float. Este método tem diversas vantagens. Além de tornar o código mais fácil de ler e manter, ele dificulta a introdução de bugs.

CONSTANTES ENUMERADAS

Podemos também definir coleções de constantes, chamadas constantes enumeradas, usando a palavra-chave *enum*. As constantes enumeradas permitem criar novos tipos e depois definir variáveis desses tipos. Os valores assumidos ficam restritos a uma determinada colecção de valores. Por exemplo, podemos declarar uma enumeração para representar os dias da semana:

```
enum DiasDaSemana
{
        Segunda,
```

```
        Terca,
        Quarta,
        Quinta,
        Sexta,
        Sabado,
        Domingo
}; // Fim de enum DiasDaSemana.
```

Depois disso, podemos definir variáveis do tipo DiasDaSemana, que somente podem assumir os valores Segunda = 0, Terca = 1, Quarta = 2, e assim por diante.

Assim, cada constante enumerada tem um valor inteiro. Se não especificarmos esse valor, a primeira constante assumirá o valor 0, a segunda constante assumirá o valor 1, e assim por diante. Se necessário, podemos atribuir um valor determinado a uma dada constante. Se somente uma constante for inicializada, as constantes subsequentes assumirão valores com incremento de 1, a partir daquela que foi inicializada. Por exemplo:

```
enum DiasDaSemana
{
        Segunda = 100,
        Terca,
        Quarta,
        Quinta,
        Sexta = 200,
        Sabado,
        Domingo
}; // Fim de enum DiasDaSemana.
```

Na declaração acima, as constantes não inicializadas assumirão os seguintes valores:

Terca = 101, Quarta = 102, Quinta = 103, Sabado = 201, Domingo = 202

As constantes enumeradas são representadas internamente como sendo do tipo int.

Exemplo

```
// Enum.cpp
// Ilustra o uso de enumerações.
#include <iostream.h>
int main()
{
// Define uma enumeração.
        enum DiasDaSemana
        {
             Segunda,
             Terca,
             Quarta,
             Quinta,
             Sexta,
             Sabado,
             Domingo
        }; // Fim de enum DiasDaSemana.
// O mesmo que:
// const int Segunda = 0; const int Terca = 1; Etc...
// const int Domingo = 6;
```

```
// Declara uma variável do tipo
// enum DiasDaSemana.
        DiasDaSemana dias;
// Uma variável int.
        int i;
        cout << "Digite um num. (0 a 6) + <Enter>:\n";
        cin >> i;
      dias = DiasDaSemana(i);
      if((dias == Sabado) || (dias == Domingo))
            cout << "Voce escolheu o fim de semana.\n";
      else
            cout << "Voce escolheu um dia util.\n";
      return 0;
} // Fim de main()
```

Saída gerada por este programa:

Digite um num. (0 a 6) + <Enter>:
5
Voce escolheu o fim de semana.

EXPRESSÕES

Em C++, uma expressão é qualquer comando (*statement*) que após ser efetuado gera um valor. Outra forma de dizer isso é: uma expressão sempre retorna um valor.

Uma expressão pode ser simples: 3.14 // Retorna o valor 3.14

Ou mais complicada: x = a + b * c / 10;

Observe que a expressão acima retorna o valor que está sendo atribuído a x. Por isso, a expressão inteira pode ser atribuída a outra variável. Por exemplo:

y = x = a + b * c / 10;

OPERADORES MATEMÁTICOS

Existem cinco operadores matemáticos em C++:

+ adição
- subtração
* multiplicação
/ divisão
% módulo

Os quatro primeiros operadores, funcionam da forma que seria de se esperar, com base na matemática elementar. O operador módulo % fornece como resultado o resto de uma divisão inteira.

OPERADORES EM PREFIXO E SUFIXO

Dois operadores muito importantes e úteis em C++ são o operador de incremento ++ e o operador de decremento --. O operador de incremento aumenta em 1 o valor da variável à qual é aplicado; o operador de decremento diminui 1. A posição dos operadores ++ e -- em relação a variável (prefixo ou sufixo) é muito importante. Na posição de prefixo, o operador é aplicado primeiro,

depois o valor da variável é acedido. Na posição de sufixo, o valor da variável é acedido primeiro, depois o operador é aplicado.

Exemplo

```
// PreSuf.cpp
// Ilustra o uso de operadores em prefixo e sufixo.
#include <iostream.h>
int main()
{
        int i = 10, j = 10;
        cout << "\n*** Valores iniciais ***\n";
        cout << "i = " << i << ", j = " << j;
// Aplica operadores.
        i++;
        ++j;
        cout << "\n*** Apos operadores ***\n";
        cout << "i = " << i << ", j = " << j;
        cout << "\n*** Exibindo usando operadores ***\n";
        cout << "i = " << i++ << ", j = " << ++j;
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

*** Valores iniciais ***
i = 10, j = 10
*** Apos operadores ***
i = 11, j = 11
*** Exibindo usando operadores ***
i = 11, j = 12

O COMANDO IF

O fluxo de execução de um programa faz com que as linhas sejam executadas na ordem em que aparecem no código. O comando *if* permite testar uma condição e seguir para uma parte diferente do código, dependendo do resultado desse teste.

A forma mais simples do comando *if* é:

if(expressão) comando;

Por exemplo:

If (a > b) a = b;

Um bloco de comandos contidos entre chaves { } tem efeito similar ao de um único comando. Portanto, o comando *if* pode ser também utilizado da seguinte forma:

If (expressao)
{
comando1;
comando2;
// etc.

}

Ou ainda,

If (expressao)
comando1;
else
comando2;

Qualquer comando pode aparecer dentro da cláusula if... else. Isso inclui até mesmo outra cláusula if... else.

Exemplo

```
// IfElse.cpp
#include <iostream.h>
int main()
{
        int numMaior, numMenor;
        cout << "Digite numMaior + <Enter>: ";
        cin >> numMaior;
        cout << "Digite numMenor + <Enter>: ";
        cin >> numMenor;
        if(numMaior >= numMenor)
        {
              if((numMaior % numMenor) == 0)
              {
                    if(numMaior == numMenor)
                          cout << "numMaior e' igual a numMenor.\n";
                    else
                          cout << "numMaior e' multiplo "
                          "de numMenor\n";
              } // Fim de if((numMaior % numMenor) == 0)
              else
                    cout << "A divisao nao e' exata.\n";
        } // Fim de if(numMaior >= numMenor)
        else
              cout << "Erro!!! numMenor e' maior " "que numMaior!\n";
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Digite numMaior + <Enter>: 44
Digite numMenor + <Enter>: 39
A divisao nao e' exata.

OPERADORES LÓGICOS

Pode ser necessário determinar se duas condições são verdadeiras ao mesmo tempo, dentro de um determinado programa. Os operadores lógicos mostrados abaixo são usados nesse tipo de situação.

Operador	Símbolo	*Exemplo*
AND	&&	expressao1 && expressao2
OR	||	expressao1 || expressao2
NOT	!	!expressao

O operador lógico AND avalia duas expressões. Se as duas forem verdadeiras, o resultado da operação lógica AND será verdadeiro. Ou seja, é preciso que ambos os lados da operação sejam verdadeiros, para que a expressão completa seja verdadeira. O operador lógico OR avalia duas expressões. Se qualquer uma delas for verdadeira, o resultado da operação lógica OR será verdadeiro. Ou seja, basta que um dos lados da operação seja verdadeiro, para que a expressão completa seja verdadeira. O operador lógico NOT avalia uma só expressão. O resultado é verdadeiro se a expressão avaliada for falsa, e vice-versa.

O OPERADOR CONDICIONAL TERNÁRIO

O operador condicional ? : é o único operador ternário de C++. Ou seja, ele recebe três termos. O operador condicional recebe três expressões e retorna um valor.

(expressao1) ? (expressao2) : (expressao3);

Esta operação pode ser interpretada da seguinte forma: se expressao1 for verdadeira, retorne o valor de expressao2; caso contrario, retorne o valor de expressao3.

Exemplo

```
// OpTern.cpp
// Ilustra o uso do operador condicional ternário.
#include <iostream.h>
int main()
{
        int a, b, c;
        cout << "Digite um num. + <Enter>: ";
        cin >> a;
        cout << "\nDigite outro num. + <Enter>: ";
        cin >> b;
        if(a == b)
              cout << "Os numeros sao iguais. " << "Tente
      novamente.\n";
        else
        {
// Atribui o valor mais alto à variável c.
              c = (a > b) ? a : b;
// Exibe os valores.
              cout << "\n*** Valores finais ***\n";
              cout << "a = " << a << "\n";
              cout << "b = " << b << "\n";
              cout << "c = " << c << "\n";
        } // Fim de else.
        return 0;
} // Fim de main()
```

Saída gerada por este programa:

Digite um num. + <Enter>: 99
Digite outro num. + <Enter>: 98
*** Valores finais ***
a = 99
b = 98
c = 99

RECURSÃO

Uma função pode chamar a si mesma. Esse processo é chamado recursão. A recursão pode ser direta ou indireta. Ela é direta quando a função chama a si mesma; na recursão indireta, uma função chama outra função, que por sua vez chama a primeira função. Alguns problemas são solucionados com mais facilidade com o uso de recursão.

O LOOP WHILE

O *loop while* faz com que o programa repita uma sequência de comandos enquanto uma determinada condição for verdadeira. A condição de teste do *loop while* pode ser complexa, desde que seja uma expressão C++ válida. Isso inclui expressões produzidas com o uso dos operadores lógicos && (AND), || (OR) e ! (NOT). Eis a forma genérica do *loop while*:

```
while(condicao)
        comando;
```

BREAK E CONTINUE

Às vezes pode ser necessário voltar para o topo do *loop while* antes que todos os comandos do *loop* tenham sido executados. O comando *continue* faz com que a execução volte imediatamente para o topo do *loop*. Outras vezes, pode ser necessário sair do loop antes que a condição de término do *loop* seja satisfeita. O comando break causa a saída imediata do *loop while*. Neste caso, a execução do programa é retomada após a chave de fechamento do loop }

O LOOP DO...WHILE

Pode acontecer do corpo de um *loop while* nunca ser executado, se a condição de teste nunca for verdadeira. Muitas vezes, é desejável que o corpo do *loop* seja executado no mínimo uma vez. Para isto, usamos o *loop do...while*. Eis a forma genérica do *loop do...while*:

```
do
        comando;
while(condicao);
```

O *loop do...while* garante que os comandos que formam o corpo do *loop* serão executados pelo menos uma vez.

O LOOP FOR

Quando usamos o *loop while*, muitas vezes precisamos definir uma condição inicial, testar essa condição para ver se continua sendo verdadeira e incrementar ou alterar uma variável a cada passagem pelo *loop*. Quando sabemos de antemão o número de vezes que o corpo do *loop* deverá ser executado, podemos usar o *loop* for, ao invés de *while*. Podemos ter *loops* for aninhados, ou seja, o corpo de um *loop* pode conter outro *loop*. O *loop* interno será executado por completo a cada passagem pelo *loop* externo.

Eis a forma genérica do loop for:
for(inicializacao; condicao; atualizacao)
comando;

O comando de inicialização é usado para inicializar o estado de um contador, ou para preparar o *loop* de alguma outra forma. A condição é qualquer expressão C++, que é avaliada a cada passagem pelo *loop*. Se condição for verdadeira, o corpo do *loop* é executado e depois a actualização é executada (geralmente, o contador é incrementado).

Exemplo

```
// LoopFor.cpp
// Ilustra o uso do loop for e while
#include <iostream.h>
int main()
{
        int contador = 0;
        cout << "\n*** Usando while ***";
        while(contador < 10)
        {
              contador++;
              cout << "\nContador = " << contador;
        } // Fim de while.
        cout << "\n\n*** Usando for ***";
        for(contador = 0; contador <= 10; contador++)
              cout << "\nContador = " << contador;
        return 0;
} // Fim de main()
```

Saída gerada por este programa:
*** Usando while ***
Contador = 1
Contador = 2
Contador = 3
Contador = 4
Contador = 5
Contador = 6
Contador = 7
Contador = 8
Contador = 9
Contador = 10
*** Usando for ***
Contador = 0
Contador = 1
Contador = 2
Contador = 3
Contador = 4
Contador = 5
Contador = 6
Contador = 7
Contador = 8
Contador = 9
Contador = 10

O COMANDO SWITCH

Já vimos como usar o comando *if...else*. O uso de *if...else* pode se tornar um tanto complicado quando existem muitas alternativas. Para essas situações, C++ oferece o comando *switch*. Eis sua forma genérica:

```
switch(expressao)
{
        case valorUm:
                comandos;
                break;
        case valorDois:
                comandos;
                break;
        ...
        case valorN:
                comandos;
                break;
        default:
                comandos;
}
```

INTRODUÇÃO A ARRAYS

Um *array* é uma coleção de elementos. Todos os elementos do *array* são obrigatoriamente do mesmo tipo de dados. Para declarar um *array*, escrevemos um tipo, seguido pelo nome do *array* e pelo índice. O índice indica o número de elementos do *array*, e fica contido entre colchetes []

Por exemplo:
long arrayLong[10];

Podemos ter *arrays* com mais de uma dimensão. Cada dimensão é representada como um índice para o *array*. Portanto, um *array* bidimensional tem dois índices; um *array* tridimensional tem três índices, e assim por diante. Os *arrays* podem ter qualquer número de dimensões. Na prática, porém, a maioria dos *arrays* têm uma ou duas dimensões.

Exemplo

```
// ArrMult.cpp
// Ilustra o uso de arrays multidimensionais.
#include <iostream.h>
int main()
{
// Declara e inicializa um
// array bidimensional.
int array2D[4][3] = { {2, 4 ,6},
{8, 10, 12},
{14, 16, 18},
{20, 22, 24}
};
// Exibe conteúdo do array.
for(int i = 0; i < 4; i++)
{
for(int j = 0; j < 3; j++)
cout << array2D[i][j]
<< "\t";
```

```
cout << "\n";
} // Fim de for(int i...
return 0;
} // Fim de main()
```

Saída gerada por este programa:
2 4 6
8 10 12
14 16 18
20 22 24

PONTEIROS

Toda variável tem um endereço. Mesmo sem saber o endereço específico de uma dada variável, podemos armazenar esse endereço em um ponteiro. Por exemplo, suponhamos que temos uma variável idade, do tipo *int*. Para declarar um ponteiro para conter o endereço dessa variável, podemos escrever:

int* pIdade = 0;

Esta linha declara *pIdade* como sendo um ponteiro para *int*. Ou seja, *pIdade* pode conter o endereço de uma variável *int*. Observe que *pIdade* é uma variável como qualquer outra. Quando declaramos uma variável inteira (do tipo *int*), ela deve pode conter um valor inteiro. Quando declaramos um variável ponteiro, como *pIdade*, ela pode conter um endereço. Assim, *pIdade* é apenas mais um tipo de variável. Neste caso, *pIdade* foi inicializada com o valor zero. Um ponteiro cujo valor é igual a zero é chamado de ponteiro nulo. Todos os ponteiros, quando são criados, devem ser inicializados com algum valor. Se não houver nenhum valor para atribuir a um ponteiro, ele deve ser inicializado com o valor zero. Um ponteiro não inicializado representa sempre um grande risco de erro.

Eis como fazer com que um ponteiro aponte para algo de útil.

```
int idade = 18;
// Uma variável inteira.
int* pIdade = 0;
// Um ponteiro para int.
pIdade = &idade;
// O ponteiro agora contém o endereço de idade.
```

A primeira linha cria uma variável, idade, do tipo *int*, e a inicializa com o valor 18. A segunda linha declara *pIdade* como sendo um ponteiro para *int*, e inicializa-o com o valor zero. O que indica que *pIdade* é um ponteiro é o asterisco * que aparece entre o tipo da variável e o nome da variável. A terceira linha atribui o endereço de idade ao ponteiro *pIdade*. A forma usada para acessar o endereço (referência) de uma variável é o operador endereço de, representado pelo caractere **&**.

Um ponteiro pode ser reutilizado quantas vezes quisermos. Para isso, basta atribuir um novo endereço ao ponteiro. Observe que neste caso, o endereço contido anteriormente no ponteiro é perdido.

Um ponteiro pode apontar para qualquer tipo de variável. Como o próprio ponteiro é uma variável, podemos fazer com que um ponteiro aponte para outro ponteiro, criando um ponteiro

para ponteiro. Veremos que este conceito é em si bastante útil. Mas é também importante para a compreensão da equivalência entre a notação de array e a notação de ponteiro.

Em C++, ponteiros e arrays estão estreitamente relacionados. Uma ponteiro para um array combina dois poderosos recursos da linguagem: a capacidade do ponteiro de proporcionar acesso indireto e a conveniência de acessar elementos de um array com o uso de índices numéricos.

Um ponteiro para um array não é muito diferente de um ponteiro para uma variável simples. Em ambos os casos, o ponteiro somente consegue apontar para um único objeto de cada vez. Um ponteiro para array, porém, pode referenciar qualquer elemento individual dentro de um array. Mas apenas um de cada vez.

Podemos usar a palavra-chave const em várias posições em relação a um ponteiro. Por exemplo, podemos declarar um ponteiro const para um valor não-const. Isso significa que o ponteiro não pode ser modificado. O valor apontado, porém, pode ser modificado.

Exemplo

```
// Ponteir.cpp
// Ilustra o uso de ponteiros.
#include <iostream.h>
int intArray[] = {10, 15, 296, 3, 18};
int main()
{
// Variáveis.
       unsigned short usVar = 200;
       unsigned long ulVar = 300;
       long lVar = 400;
// Ponteiros.
       unsigned short* usPont = &usVar;
       unsigned long* ulPont = &ulVar;
       long* lPont = &lVar;
       cout << "\n*** Valores iniciais ***\n";
       cout << "\nusVar: Valor = "<< usVar << ", Endereco = "
<< usPont;
       cout << "\nulVar: Valor = "<< ulVar << ", Endereco = "
<< ulPont;
       cout << "\nlVar: Valor = "<< lVar << ", Endereco = "
<< lPont << "\n";
// Modifica valores das variáveis
// usando os ponteiros.
       *usPont = 210;
       *ulPont = 310;
       *lPont = 410;
// Exibe os novos valores.
       cout << "\n*** Novos valores ***\n";
       cout << "\nusVar: Valor = "<< usVar << ", Endereco = "
<< usPont;
       cout << "\nulVar: Valor = "<< ulVar << ", Endereco = "
<< ulPont;
       cout << "\nlVar: Valor = "<< lVar << ", Endereco = "
<< lPont << "\n";
// Uma variável int.
       int iVar = 1000;
```

```
// Um ponteiro para int,inicializado com o endereço de iVar.
	int *iPtr = &iVar;
// Um ponteiro para ponteiro para int, inicializado com o
// endereço de iPtr.
	int **iPtrPtr = &iPtr;
// Exibe valor da variável,acessando-o via iPtr.
	cout << "Acessando valor via ponteiro\n";
	cout << "iVar = "<< *iPtr<< "\n";
// Exibe valor da variável,acessando-o via iPtrPtr.
	cout << "Acessando valor via ponteiro para ponteiro\n";
	cout << "iVar = "<< **iPtrPtr<< "\n";
// Declara um ponteiro para int.
	int *arrPont;
// Uma variável int.
	int i;
// Atribui ao ponteiro o endereço do primeiro elemento do array.
	arrPont = &intArray[0];
// Exibe os elementos do array,acessando-os via ponteiro.
	for(i = 0; i < 5; i++)
	{
		cout << "intArray["<< i<< "] = "<< *arrPont<< "\n";
// Incrementa o ponteiro.
		arrPont++;
	} // Fim de for.
	return 0;
} // Fim de main()
```

Saída gerada por este programa:

```
*** Valores iniciais ***
usVar: Valor = 200, Endereco = 0066FE02
ulVar: Valor = 300, Endereco = 0066FDFC
lVar: Valor = 400, Endereco = 0066FDF8
*** Novos valores ***
usVar: Valor = 210, Endereco = 0066FE02
ulVar: Valor = 310, Endereco = 0066FDFC
lVar: Valor = 410, Endereco = 0066FDF8
Acessando valor via ponteiro
iVar = 1000
Acessando valor via ponteiro para ponteiro
iVar = 1000
intArray[0] = 10
intArray[1] = 15
intArray[2] = 296
intArray[3] = 3
intArray[4] = 18
```

REFERÊNCIAS

Referência é um modificador que permite a criação de novos tipos derivados. Assim como pode-se criar um tipo ponteiro para um inteiro, pode-se criar uma referência para um inteiro. A declaração de uma referência é análoga à de um ponteiro, usando o caracter **&** no lugar de *****.

Uma referência para um objeto qualquer é, internamente, um ponteiro para o objeto. Mas, diferentemente de ponteiros, uma variável que é uma referência é utilizada como se fosse o

próprio objeto. Os exemplos deixarão estas ideias mais claras. Vamos analisar referências em três utilizações: como variáveis locais, como tipos de parâmetros e como tipo de retorno de funções.

Uma variável local que seja uma referência deve ser sempre inicializada; a não inicialização causa um erro de compilação. Como as referências se referenciam a objetos, a inicialização não pode ser feita com valores constantes:

```
{
        int a; // ok, variável normal
        int& b = a; // ok, b é uma referencia para a
        int& c; // erro! não foi inicializada
        int& d = 12; // erro! inicialização inválida
}
```

A variável b é utilizada como se fosse realmente um inteiro, não há diferença pelo fato de ela ser uma referência. Só que b não é um novo inteiro, e sim uma referência para o inteiro guardado em a. Qualquer alteração em a se reflete em b e vice-versa. É como se b fosse um novo nome para a mesma variável a.

No caso de a referência ser um tipo de algum argumento de função, o parâmetro será passado por referência:

```
void f(int a1, int &a2, int *a3)
{
        a1 = 1; // altera cópia local
        a2 = 2; // altera a variável passada (b2 de main)
        *a3 = 3; // altera o conteúdo do endereço de b3
}
void main()
{
        int b1 = 10, b2 = 20, b3 = 30;
        f(b1, b2, &b3);
        printf("b1=%d, b2=%d, b3=%d\n", b1, b2, b3);
        // imprime b1=10, b2=2, b3=3
}
```

Falta ainda analisar um outro uso de referências, quando esta aparece como tipo de retorno de uma função. Por exemplo:

```
int& f()
{
        static int global;
        return global; // retorna uma referência para a variável
}
void main()
{
        f() = 12; // altera a variável global
}
```

É importante notar que este exemplo é válido porque global é uma variável *static* de f, ou seja, é uma variável global com âmbito limitado a f. Se global fosse uma variável local comum, o valor de retorno seria inválido, pois quando a função f terminasse a variável global não existiria mais, e portanto a referência seria inválida.

Programação Orientada por Objetos – Tópico 3

***Temática**: Definição e criação de classes e seus atributos*

***Actividade 3**: Familiarização com o conceito de classe, objeto e seus atributos

Competências a desenvolver:

- Compreender o conceito de classe e objeto
- Aprender os diferentes níveis de controle de acesso
- Compreender a utilização de classes amigas e aninhadas
- Aprender a utilizar os construtores e destrutores

Classe em sentido lato: tipo de dados definido pelo utilizador (programador) [Stroustrup]

- inclui enumerações (enum), uniões (union), estruturas (struct) e classes em sentido estrito (class)
- tipos de dados definidos em bibliotecas standard são classes
- tipos de dados built-in ou construídos com apontadores, arrays ou referências (mesmo que nomeados com typedef) não constituem classes

Classe em sentido estrito: tipo de dados definido com class

- é uma generalização do conceito de estrutura em C
- Para além de dados (membros-dados), uma classe pode também conter funções de manipulação desses dados (membros-funções), restrições de acesso a ambos (dados e funções) e redefinições de quase todos os operadores de C++ para objectos da classe

Controle de acesso - public e private

Parte do objetivo de uma classe é esconder o máximo de informação possível. Então é necessário impor certas restrições na maneira como uma classe pode ser manipulada, e que dados e código podem ser usados. Podem ser estabelecidos três níveis de permissão de acordo com o contexto de uso dos atributos:

- nos métodos da classe
- a partir de objetos da classe
- métodos de classes derivadas (este ponto será visto posteriormente)

Cada um destes três contextos tem privilégios de acesso diferenciados; cada um tem uma palavra reservada associada, *private*, *public* e *protected* respectivamente. O exemplo abaixo ilustra o uso destas novas palavras reservadas:

```
struct controle {
     private:
            int a;
            int f1( char* b );
     protected:
            int b;
            int f2( float );
     public:
            int c;
            float d;
            void f3( controle* );
};
```

As seções podem ser declaradas em qualquer ordem, inclusive podem aparecer mais de uma vez. Atributos *private* são os mais restritos. Somente a própria classe pode acessar os atributos privados. Ou seja, somente os métodos da própria classe tem acesso a estes atributos. Os atributos declarados como *public*, podem ser acedidos através da instanciação da classe (o objeto). Membros protected serão explicados quando forem introduzidas as classes derivadas.

Para exemplificar melhor o uso do controle de acesso, vamos considerar uma implementação de um conjunto de inteiros. As operações necessárias são, por exemplo, inserir e retirar um elemento, verificar se um elemento pertence ao conjunto e a cardinalidade do conjunto. Então o nosso conjunto terá pelo menos o seguinte:

```
struct Conjunto {
       void insere(int n);
       void retira(int n);
       int pertence(int n);
       int cardinalidade();
};
```

Se a implementação deste conjunto usar listas encadeadas, é preciso usar uma estrutura auxiliar elemento que seriam os nós da lista. A nova classe teria ainda uma variável *private* que seria o ponteiro para o primeiro elemento da lista. Outra variável *private* seria um contador de elementos. Vamos acrescentar ainda um método para limpar o conjunto, para ser usado antes das funções do conjunto propriamente ditas. Eis a definição da classe:

```
struct Conjunto {
     private:
            struct listElem {
                 listElem *prox;
                 int valor;
            };
            listElem* lista;
            int nElems;
       public:
            void limpa() { nElems=0; lista=NULL; }
            void insere(int n);
            void retira(int n);
            int pertence(int n);
            int cardinalidade();
};
```

Como somente os métodos desta classe tem acesso aos campos privados, a estrutura interna não pode ser alterada por quem usa o conjunto, o que garante a consistência do conjunto.

Declaração de classes com a palavra reservada class

As classes podem ser declaradas usando-se a palavra reservada *class* no lugar de *struct*. A diferença entre as duas opções é o nível de proteção caso nenhum dos especificadores de acesso seja usado. Os membros de uma *struct* são públicos, enquanto que em uma *class*, os membros são privados:

```
struct A {
       int a; // a é público
};
```

```
class B {
        int a; // a é privado
};
```

Como as interfaces das classes devem ser as menores possíveis e devem ser também explícitas, as declarações com a palavra class são mais usadas. Com *class*, somente os nomes explicitamente declarados como *public* são exportados. A partir de agora os exemplos vão ser feitos usando-se a palavra *class*.

Classes e funções friend

Algumas vezes duas classes são tão próximas conceitualmente que seria desejável que uma delas tivesse acesso irrestrito aos membros da outra. No exemplo anterior, não há meio de percorrer o conjunto para, por exemplo, imprimir todos os elementos. Para fazer isto, seria preciso ter acesso ao dado lista e à estrutura *listElem*, ambos *private*. Uma maneira de resolver este problema seria aumentar a interface do conjunto oferecendo funções para percorrer os elementos.

A solução normalmente usada é a idéia de iteradores. Um iterador atua sobre alguma coleção de elementos e a cada chamada retorna um elemento diferente da coleção, até que já tenha retornado todos. Um iterador para o nosso conjunto seria:

```
class IteradorConj {
        Conjunto::listElem *corrente;
      public:
        void inicia( Conjunto* c ) { corrente = c->lista; }
        int terminou() { return corrent==NULL; }
        int proxElem() { int n=corrente->valor; corrent=corrente-
>prox; return n; }
};
void main()
{
        Conjunto conj; // cria conjunto
        conj.limpa(); // inicializa para operações
        conj.insere(10); //insere o elemento 10
// ...
        IteradorConj it; // cria um iterador
        it.inicia( &conj ); // inicializa o iterador com conj
        while (!it.terminou()) // percorre todos os elementos
             printf("%d\n", it.proxElem() ); // imprimindo-os
}
```

O problema aqui é que o iterador usa dados privados de *Conjunto*, o que gera erros durante a compilação. Esse é um caso em que as duas classes estão intimamente ligadas, e então seria conveniente, na declaração de *Conjunto*, dar acesso irrestrito à classe *IteradorConj*.

Para isso existe a palavra reservada *friend*. Ela serve para oferecer acesso especial à algumas classes ou funções. Na declaração da classe Conjunto é possível dar permissão irrestrita aos seus atributos. A classe *Conjunto* chega à sua forma final:

```
class Conjunto {
        friend class IteradorConj;
```

```
        struct listElem {
              ...
              ...
```

Também é possível declarar funções *friend*. Nesse caso, a função terá acesso irrestrito aos componentes da classe que a declarou como *friend*. Exemplo de função *friend*:

```
class No {
        friend int leValor( No* ); // dá acesso privilegiado
        // à função leValor
        int valor;
        public:
        void setaValor( int v ) { valor=v; }
};
int leValor( No* n )
{
        return n->valor; // acessa dado private de No
}
```

O recurso de classes e funções *friend* devem ser usados com cuidado, pois isto é um furo no encapsulamento dos dados de uma classe. Projetos bem elaborados raramente precisam lançar mão de classes ou funções *friend*.

Construtores e destrutores

Como o nome já indica, um construtor é uma função usada para construir um objeto de uma dada classe. Ele é chamado automaticamente assim que um objeto é criado. Analogamente, os destrutores são chamados assim que os objetos são destruídos. Eles servem para garantir a consitência do programa e evitar erros na gestão da memória e recursos.

Assim como o controle de acesso desempenha um papel importante para manter a consistência interna dos objetos, os construtores são fundamentais para garantir que um objeto recém criado esteja também consistente. Destrutores são normalmente utilizados para liberar recursos alocados pelo objeto, como memória, arquivos etc.

Declaração de construtores e destrutores

Os construtores e destrutores são métodos especiais. Nenhum dos dois tem um tipo de retorno, e o destrutor não pode receber parâmetros, ao contrário do construtor. A declaração de um construtor é feita definindo-se um método com o mesmo nome da classe. O nome do destrutor é o nome da classe precedido de ~ (til). Ou seja, um construtor de uma classe X é declarado como um método de nome X, o nome do destrutor é ~X; ambos sem tipo de retorno. O destrutor não pode ter parâmetros; o construtor pode. Pelo mecanismo de sobrecarga (funções são diferenciadas não apenas pelo nome, mas pelos parâmetros também), classes podem ter mais de um construtor.

Com o uso de construtores, as classe Conjunto e IteradorConj passa a ser:

```
class Conjunto {
        friend class IteradorConj;
        struct listElem {
              listElem *prox;
              int valor;
        };
        listElem* lista;
```

```
        int nElems;
      public:
        Conjunto() { nElems=0; lista=NULL; }
        ~Conjunto(); // desaloca os nós da lista
        void insere(int n);
        void retira(int n);
        int pertence(int n);
        int cardinalidade();
};
class IteradorConj {
        Conjunto::listElem *corrente;
      public:
        IteradorConj( Conjunto* c ) { corrente = c->lista; }
        int terminou() { return corrent==NULL; }
        int proxElem()
        { int n=corrente->valor; corrent=corrente->prox; return n; }
};
```

Chamada de construtores e destrutores

Os construtores são chamados automaticamente sempre que um objeto da classe for criado. Ou seja, quando a variável é declarada (objetos alocados na pilha) ou quando o objeto é alocado com new (objetos dinâmicos alocados no *heap*).

Os destrutores de objetos alocados na pilha são chamados quando o objeto sai do seu âmbito. O destrutor de objetos alocados com *new* só é chamado quando estes são desalocados com *delete*.

```
class A {
  public:
        A() { printf("construtor\n"); }
        ~A() { printf("destrutor\n"); }
};
void main()
{
        A a1; // chama construtor de a1
        {
              A a2; // chama construtor de a2
              A *a3 = new A; // chama construtor de a3
              delete a3; // chama destrutor de a3
        } // chama destrutor de a2

} // chama destrutor de a1
```

Construtores com parâmetros

No exemplo sobre conjuntos, a classe *IteradorConj* possui um construtor que recebe um parâmetro. Se os construtores existem para garantir a consistência inicial dos objetos, o código está indicando que, para que um iterador esteja consistente, é necessário fornecer um conjunto sobre o qual será feita a iteração.

Se for possível criar um *IteradorConj* sem fornecer este conjunto, então o construtor não está garantindo nada. Mas não é isto que acontece. A declaração de um construtor impõe que os objetos só sejam criados através deles. Sendo assim, não é mais possível criar um *IteradorConj* sem fornecer um *Conjunto*.

Construtores gerados automaticamente

O fato de algumas classes não declararem construtores não significa que elas não tenham construtores. Na realidade, o compilador gera alguns construtores automaticamente. Um dos construtores só é gerado se a classe não declarar nenhum. Este é o construtor vazio, que permite que os objetos sejam criados. Por exemplo, como a classe

```
class X {
        int a;
};
```

não declara nenhum construtor, o compilador automaticamente gera um construtor vazio público para esta classe. A declaração abaixo é equivalente:

```
class X {
        int a;
        public: X() {} // construtor vazio
};
```

Outro construtor gerado é o construtor de cópia. Este é gerado mesmo que a classe declare algum outro construtor. O construtor de cópia de uma classe recebe como parâmetro uma referência para um objeto da própria classe. A classe X acima possui este construtor:

```
class X {
        int a;
        // public: X() {} construtor vazio
        // public: X(const X&); construtor de cópia
};
```

O construtor de cópia constrói um objeto a partir de outro do mesmo tipo. O novo objeto é uma cópia byte a byte do objeto passado como parâmetro. Repare que a existência deste construtor não é um furo na consistência dos objetos, já que ele só pode ser usado a partir de um objeto existente, e portanto, consistente. Este construtor pode ser chamado de duas formas:

```
void main()
{
        X a1; // usa construtor vazio
        X a2(a1); // usa construtor de cópia
        X a3 = a2; // usa construtor de cópia
}
```

A atribuição só chama o construtor de cópia quando usada junto com a declaração do objeto. É importante notar que este construtor pode ser redefinido, basta declarar um construtor com a mesma assinatura (ou seja, recebendo como parâmetro uma referência para um objeto da própria classe).

Objetos temporários

Assim como não é preciso declarar uma variável dos tipos primitivos sempre que se quer usar um valor temporariamente, é possível criar objetos temporários em C++. Uma utilização típica é na passagem de objetos como parâmetro de funções.

Conversão por construtores

Um construtor com apenas um parâmetro pode ser visto como uma função de conversão do tipo do parâmetro para o tipo da classe. Por exemplo,

```
class A {
        public:
              A(int);
              A(char*, int = 0);
};
void f(A);
void main()
{
        A a1 = 1; // a1 = A(1)
        A a2 = "abc"; // a2 = A("abc", 0)
        a1 = 2; // a1 = A(2)
        f(3); // f(A(3))
}
```

Construtores privados

Os construtores, assim como qualquer método, podem ser privados. Como o construtor é chamado na criação, os objetos só poderão ser criados com este construtor dentro de métodos da própria classe ou em classes e funções *friend*.

Destrutores privados

Destrutores também podem ser privados. Isto significa que objetos desta classe só podem ser destruídos onde os destrutores podem ser chamados (métodos da própria classe, classes e funções *friend*). Usando este recurso, é possível projetar classes cujos objetos não são nunca destruídos. Outra possibilidade é o projeto de objetos que não podem ser alocados na pilha, apenas dinamicamente. Para permitir a destruição de objetos terá que ser incluído um método na classe que execute um *delete this*.

Exemplo:

```
class A {
        ~A() {}
        public:
              int a;
};
void main()
{
        A a1; // erro! destrutor privado,não pode ser chamado
        // quando o objeto sair do âmbito
        A* a2 = new A; // ok, só não pode usar delete depois
}
```

Inicialização de campos de classes com construtores

Quando um objeto não tem um construtor sem parâmetros, é preciso passar obrigatoriamente valores como os parâmetros. No caso do objeto ser uma variável, basta definir os parâmetros na hora da declaração:

```
class A {
        public: A(int);
};
A a(123);
```

Mas e se o objeto for um campo de uma outra classe? Nesse caso ele estará sendo criado quando um objeto desta outra classe for criado, e nessa hora os parâmetros precisarão estar disponíveis:

```
class A {
        public: A(int);
};
class B {
        A a;
};
```

Ao se criar um objeto do tipo B, que inteiro deve ser passado ao campo a? Nesse caso, o construtor de B tem que especificar este inteiro, e o compilador não gera um construtor vazio. Ou seja, a classe B tem que declarar um construtor para que seja possível criar objetos deste tipo. A sintaxe é a seguinte:

```
class B {
        A a;
        public:
              B() : a(3) {}
};
```

Métodos const

Quando declaramos um método como sendo *const*, estamos indicando que esse método não alterará o valor de nenhum dos membros da classe. Para declarar um método de classe como sendo const, colocamos esta palavra-chave após os parênteses ( ), mas antes do caractere de ponto e vírgula ; Em geral, os métodos de acesso são declarados como *const*.

Quando declaramos uma função como const, qualquer tentativa que façamos na implementação dessa função de alterar um valor do objeto será sinalizada pelo compilador como sendo um erro. Isso faz com que o compilador detecte erros durante o processo de desenvolvimento, evitando a introdução de bugs no programa.

Objetos como membros

Muitas vezes, construimos uma classe complexa declarando classes mais simples e incluindo objetos dessas classes simples na declaração da classe mais complicada. Por exemplo, poderíamos declarar uma classe roda, uma classe motor, uma classe transmissão, e depois combinar objetos dessas classes para criar uma classe carro, mais complexa. Chamamos esse tipo de relacionamento tem-um, porque um objeto tem dentro de si outro objeto. Um carro tem um motor, quatro rodas e uma transmissão.

Exemplo

```
//-----------------------------------------------------------------
// ObjMemb.h
// Ilustra uma classe que tem objetos de outra classe como membros.
#include <iostream.h>
class Ponto
{
        int coordX;
        int coordY;
        public:
```

```
            void defineX(int vlrX)
            {
                  coordX = vlrX;
            } // Fim de defineX()
            void defineY(int vlrY)
            {
                  coordY = vlrY;
            } // Fim de defineY()
            int acessaX() const
            {
                  return coordX;
            } // Fim de acessaX()
            int acessaY() const
            {
                  return coordY;
            } // Fim de acessaY()
}; // Fim de class Ponto.
// No sistema de coords considerado, a origem (0, 0) fica no canto
// superior esquerdo.
class Retangulo
{
        Ponto supEsq;
        Ponto infDir;
        public:
        // Construtor.
            Retangulo(int esq, int topo,int dir, int base);
       // Destrutor.
            ~Retangulo()
            {
                  cout << "Destruindo retangulo...";
            } // Fim de ~Retangulo()
      // Funções de acesso.
            Ponto acessaSupEsq() const
            {
                  return supEsq;
            } // Fim de acessaSupEsq() const
            Ponto acessaInfDir() const
            {
                  return infDir;
            } // Fim de acessainfDir() const
      // Funções para definir valores.
            void defineSupEsq(Ponto se)
            {
                  supEsq = se;
            } // Fim de defineSupEsq()
            void defineInfDir(Ponto id)
            {
                  infDir = id;
            } // Fim de defineInfDir()
        // Função para calcular a área do retângulo.
            int calcArea() const;
}; // Fim de class Retangulo.
//--------------------------------------------------------------
//--------------------------------------------------------------
// ObjMemb.cpp
// Ilustra uma classe que tem objetos de outra classe como membros.
```

```
#include "ObjMemb.h"
// Implementações.
Retangulo::Retangulo(int esq, int topo,int dir, int base)
{
        supEsq.defineY(topo);
        supEsq.defineX(esq);
        infDir.defineY(base);
        infDir.defineX(dir);
} // Fim de Retangulo::Retangulo()
int Retangulo::calcArea() const
{
        int xd, xe, ys, yi;
        xd = infDir.acessaX();
        xe = supEsq.acessaX();
        yi = infDir.acessaY();
        ys = supEsq.acessaY();
        return (xd - xe) * (yi - ys);
} // Fim de Retangulo::calcArea() const
int main()
{
// Solicita coords para o retangulo.
        int xse, yse, xid, yid;
        cout << "\nDigite x sup. esq.: ";
        cin >> xse;
        cout << "\nDigite y sup. esq.: ";
        cin >> yse;
        cout << "\nDigite x inf. dir.: ";
        cin >> xid;
        cout << "\nDigite y inf. dir.: ";
        cin >> yid;
// Cria um objeto da classe Retangulo.
        Retangulo ret(xse, yse, xid, yid);
        int areaRet = ret.calcArea();
        cout << "\nArea do retangulo = "<< areaRet<< "\n";
} // Fim de main()
```

Classes internas

Vimos que C++ permite que uma classe contenha objetos de outra classe. Além disso, C++ permite também que uma classe seja declarada dentro de outra classe. Dizemos que a classe interna está aninhada dentro da classe externa.

Exemplo

```
//-------------------------------------------
// ClMemb.cpp
// Ilustra uma classe como membro de outra classe.
#include <iostream.h>
class Externa
{
        int dadoExt;
        public:
        // Construtor.
             Externa()
             {
                  cout << "\nConstruindo obj. Externa...";
                  dadoExt = 25;
```

```
		} // Fim do construtor Externa()
		// Destrutor.
		~Externa()
		{
			cout << "\nDestruindo obj. Externa...";
		} // Fim do destrutor ~Externa()
// Uma classe aninhada.
		class Interna
		{
			int dadoInt;
			public:
			// Construtor.
				Interna()
				{
					cout <<"\nConstruindo obj interna...";
					dadoInt = 50;
				} // Fim do constr. Interna()
		// Destrutor.
				~Interna()
				{
					cout << "\nDestruindo obj Interna...";
				} // Fim do destr. ~Interna()
				// Um método public.
				void mostraDadoInt()
				{
					cout << "\ndadoInt = "<< dadoInt;
				} // Fim de mostraDadoInt()
		} objInt; // Um objeto de class Interna.
// Um método public.
		void mostraTudo()
		{
			objInt.mostraDadoInt();
			cout << "\ndadoExt = "<< dadoExt;
		} // Fim de mostraTudo()
}; // Fim de class Externa.
int main()
{
// Um objeto de class Externa.
	Externa objExt;
	objExt.mostraTudo();
} // Fim de main()
//---------------------------------------------
```

Ponteiros como membros de uma classe

Um ou mais membros de uma classe podem ser ponteiros para objetos do free store. A memória pode ser alocada no construtor da classe, ou em um de seus métodos, e pode ser liberada no construtor.

O ponteiro this

Toda função membro de classe tem um parâmetro oculto: o ponteiro this. O ponteiro this aponta para o próprio objeto. Portanto, em cada chamada a uma função membro, ou em cada acesso a um membro de dados, o ponteiro this é passado como um parâmetro oculto. Veja o exemplo.

Exemplo

```
//-------------------------------------------------------
// ThisPnt.cpp
// Ilustra o uso do ponteiro this.
#include <iostream.h>
// Define uma classe.
class Cliente
{
	int idCliente;
	float saldo;
	public:
// Construtor.
		Cliente(int id, float sal)
		{
			cout << "\nConstruindo obj. Cliente...\n";
			this->idCliente = id;
			this->saldo = sal;
		} // Fim de Cliente()
// Destrutor.
		~Cliente()
		{
			cout << "\nDestruindo obj. Cliente "<< this-
      >idCliente<< " ...\n";
		} // Fim de ~Cliente()
// Um método para exibir valores do cliente.
		void mostraCliente() const
		{
			cout << "\nidCliente = "<< this->idCliente
<< "\tSaldo = "<< this->saldo;
		} // Fim de mostraCliente()
// Métodos para alterar os valores do cliente.
		void defineId(int id)
		{
			this->idCliente = id;
		} // Fim de defineId()
		void defineSaldo(float sal)
		{
			this->saldo = sal;
		} // Fim de defineSaldo()
}; // Fim de class Cliente.
int main()
{
// Um ponteiro para Cliente.
	Cliente* pCliente;
// Cria um objeto Cliente com new.
	cout << "\nCriando Cliente com new...";
	pCliente = new Cliente(10, 25.0);
// Checa se alocação foi bem sucedida.
	if(pCliente == 0)
	{
		cout << "\nErro alocando memoria...\n";
		return 1;
	} // Fim de if.
// Cria uma variável local Cliente.
	cout << "\nCriando Cliente local...";
	Cliente clienteLocal(20, 50.0);
// Exibe os objetos.
```

```
        pCliente->mostraCliente();
// O mesmo que:(*pCliente).mostraCliente();
        clienteLocal.mostraCliente();
// Altera valores.
        cout << "\nAlterando valores...";
        pCliente->defineId(40);
        pCliente->defineSaldo(400.0);
        clienteLocal.defineId(80);
        clienteLocal.defineSaldo(800.0);
// Exibe os novos valores dos objetos.
        cout << "\n\nNovos valores...";
        pCliente->mostraCliente();
        clienteLocal.mostraCliente();
// Deleta Cliente dinâmico.
        delete pCliente;
        return 0;
} // Fim de main()
//---------------------------------------------------
```

Referências a objetos

Podemos ter referência a qualquer tipo de variável, inclusive variáveis de tipos definidos pelo utilizador. Observe que podemos criar uma referência a um objeto, mas não a uma classe.

Não podemos inicializar uma referência com a classe Cliente:

```
Cliente & refCliente = Cliente; // Erro!!!
```

Precisamos inicializar refCliente com um objeto Cliente em particular.

```
Cliente fulano;
Cliente & refCliente = fulano;
```

Programação Orientada por Objetos – Tópico 4

***Temática**: Sobrecarga, conversão e arrays de objetos*

Actividade 4: Familiarização com o conceito de sobrecarga e conversão, utilização de arrays de objetos

Competências a desenvolver:

- Compreender o conceito de sobrecarga
- Aprender a criar sobrecarga em funções membro e operadores
- Aprender a fazer a conversão entre objetos e tipos simples
- Aprender a trabalhar com arrays e listas de objetos

Funções membro sobrecarregadas

No tópico 2 vimos que podemos implementar a sobrecarga de funções, escrevendo duas ou mais funções com o mesmo nome, mas com diferentes listas de parâmetros. As funções membros de classes podem também ser sobrecarregadas, de forma muito similar, conforme mostrado no exemplo abaixo.

```
//-----------------------------------------------
// SbrMemb.cpp
// Ilustra sobrecarga de funções membro.
#include <iostream.h>
class Retangulo
{
        int altura;
        int largura;
        public:
// Construtor.
              Retangulo(int alt, int larg);
// Função sobrecarregada.
              void desenha();
              void desenha(char c);
}; // Fim de class Retangulo.
// Implementação.
// Construtor.
Retangulo::Retangulo(int alt, int larg)
{
        altura = alt;
        largura = larg;
} // Fim de Retangulo::Retangulo()
// Função sobrecarregada.
void Retangulo::desenha()
// Desenha o retângulo preenchendo-o com o caractere '*'
{
        for(int i = 0; i < altura; i++)
        {
              for(int j = 0; j < largura; j++) cout << '*';
              cout << "\n";
        } // Fim de for(int i = 0...
} // Fim de Retangulo::desenha()
void Retangulo::desenha(char c)
// Desenha o retângulo preenchendo-o
// com o caractere c recebido como argumento.
{
        for(int i = 0; i < altura; i++)
        {
              for(int j = 0; j < largura; j++)
```

```
                    cout << c;
              cout << "\n";
        } // Fim de for(int i = 0...
} // Fim de Retangulo::desenha(char c)
int main()
{
// Cria um objeto da classe Retangulo.
        Retangulo ret(8, 12);
// Desenha usando as duas versões de desenha()
        ret.desenha();
        cout << "\n\n";
        ret.desenha('A');
        return 0;
} // Fim de main()
//---------------------------------------------------
```

Funções membro com valores default

Da mesma forma que as funções globais podem ter valores default, o mesmo acontece com funções membro de uma classe. O exemplo abaixo ilustra esse fato.

```
//-------------------------------------------------------
// MembDef.cpp
// Ilustra uso de valores default em funções membro.
#include <iostream.h>
class Retangulo
{
        int altura;
        int largura;
        public:
// Construtor.
              Retangulo(int alt, int larg);
// Função com valor default.
              void desenha(char c = '*');
}; // Fim de class Retangulo.
// Implementação.
// Construtor.
Retangulo::Retangulo(int alt, int larg)
{
        altura = alt;
        largura = larg;
} // Fim de Retangulo::Retangulo()
// Função com valor default.
void Retangulo::desenha(char c)
// Desenha o retângulo preenchendo-o com o caractere c
{
        for(int i = 0; i < altura; i++)
        {
              for(int j = 0; j < largura; j++)
                    cout << c;
              cout << "\n";
        } // Fim de for(int i = 0...
} // Fim de Retangulo::desenha()
int main()
{
// Cria um objeto da classe Retangulo.
        Retangulo ret(8, 12);
// Desenha usando valor default.
        ret.desenha();
        cout << "\n\n";
```

```
// Desenha especificando caractere.
        ret.desenha('C');
        return 0;
} // Fim de main()
//----------------------------------------
```

Sobrecarregando construtores

Os construtores, tal como as outras funções membro, podem ser sobrecarregados. A possibilidade de sobrecarregar construtores representa um recurso poderoso e flexível. Por exemplo, podemos ter uma classe Retangulo com dois construtores: o primeiro recebe argumentos para a altura e a largura; o segundo não recebe nenhum argumento, construindo um retângulo com um tamanho padrão.

Inicializando variáveis membro

Até agora, temos definido os valores das variáveis membro dentro do corpo do construtor. Porém os construtores são chamados em dois estágios: o estágio de inicialização e o corpo da função.
A maioria das variáveis pode ter seus valores definidos em qualquer dos dois estágios. Porém é mais eficiente, e mais elegante, inicializar as variáveis membro no estágio de inicialização. O exemplo abaixo ilustra como isso é feito.

```
//----------------------------------------------------------
// InicVar.cpp
// Ilustra inicialização de variáveis membro.
#include <iostream.h>
class Retangulo
{
        int altura;
        int largura;
        public:
// Construtores sobrecarregados.
// Default.
        Retangulo();
        Retangulo(int alt, int larg);
// Função com valor default.
        void desenha(char c = '*');
}; // Fim de class Retangulo.
// Implementação.
// Construtor default.
        Retangulo::Retangulo() : altura(7), largura(11)
{
        cout << "\nConstrutor default...\n";
} // Fim de Retangulo::Retangulo()
Retangulo::Retangulo(int alt, int larg) : altura(alt), largura(larg)
{
        cout << "\nConstrutor (int, int)...\n";
} // Fim de Retangulo::Retangulo(int, int)
// Função com valor default.
void Retangulo::desenha(char c)
// Desenha o retângulo preenchendo-o com o caractere c
{
        for(int i = 0; i < altura; i++)
        {
              for(int j = 0; j < largura; j++)
                    cout << c;
              cout << "\n";
} // Fim de for(int i = 0...
```

```
} // Fim de Retangulo::desenha()
int main()
{
// Cria um objeto da classe Retangulo
// usando construtor default.
        Retangulo ret1;
// Cria outro objeto especificando as dimensões.
        Retangulo ret2(8, 12);
// Desenha ret1.
        ret1.desenha('1');
        cout << "\n\n";
// Desenha ret2.
        ret2.desenha('2');
        return 0;
} // Fim de main()
//------------------------------------------------------
```

Sobrecarga de operadores: os que podem ser redefinidos

A maior parte dos operadores podem ser sobrecarregados. São eles:

new delete

+ - * / % ^& | ~

! = < > += -= *= /= %=

^= &= |= << >> >>= <<= == !=

<= >= && || ++ -- , ->* ->

() []

Tanto as formas unárias como as binárias de

+ - * &

podem ser sobrecarregadas, assim como as formas pré-fixadas ou pós fixadas de

++ --

Os seguintes operadores não podem ser sobrecarregados:

. .* :: sizeof ?:

já que estes operadores já têm um significado predefinido (exceto ?:) para objetos de qualquer classe.

A função de atribuição operator=() é definida por default para todos os objetos como a atribuição byte a byte dos campos do objeto.

Sobrecarregando o operador ++

Cada um dos tipos embutidos de C++, como int, float e char, tem diversos operadores que se aplicam a esse tipo, como o operador de adição (+) e o operador de multiplicação (*). C++ permite que o programador crie também operadores para suas próprias classes, utilizando a sobrecarga de operadores.

Para ilustrar o uso da sobrecarga de operadores, começaremos criando uma nova classe, chamada Contador. Um objeto Contador poderá ser usado em loops e outras situações nas quais um número deve ser incrementado, decrementado e ter seu valor acessado.

Por enquanto, os objetos de nossa classe Contador não podem ser incrementados, decrementados, somados, atribuídos nem manuseados de outras formas. Nos próximos passos, acrescentaremos essa funcionalidade a nossa classe.

```
//--------------------------------------------------------
// SobreO.cpp
// Ilustra sobrecarga de operadores.
#include <iostream.h>
// Declara a classe Contador.
class Contador
```

```
{
        unsigned int vlrCont;
        public:
// Construtor.
              Contador();
// Destrutor.
              ~Contador();
              unsigned int acessaVal() const;
              void defineVal(unsigned int val);
}; // Fim de class Contador.
// Implementação.
// Construtor.
Contador::Contador() : vlrCont(0)
{
        cout << "\nConstruindo Contador...\n";
} // Fim de Contador::Contador()
// Destrutor.
Contador::~Contador()
{
        cout << "\nDestruindo Contador...\n";
} // Fim de Contador::~Contador()
unsigned int Contador::acessaVal() const
{
        return vlrCont;
} // Fim de Contador::acessaVal()
void Contador::defineVal(unsigned int val)
{
        vlrCont = val;
} // Fim de Contador::defineVal()
int main()
{
// Um objeto contador.
        Contador umCont;
// Exibe valor.
        cout << "\nValor de contador = "<< umCont.acessaVal();
        return 0;
} // Fim de main()
//--------------------------------------------------
```

Podemos acrescentar a possibilidade de incrementar um objeto Contador de duas maneiras. A primeira é escrever um método incrementar(). Esse método é ilustrado no exemplo abaixo.

```
//--------------------------------------------------
// SobreO2.cpp
// Ilustra sobrecarga de operadores.
// Acrescenta função incrementar()
#include <iostream.h>
// Declara a classe Contador.
class Contador
{
        unsigned int vlrCont;
        public:
// Construtor.
              Contador();
// Destrutor.
              ~Contador();
              unsigned int acessaVal() const;
              void defineVal(unsigned int val);
              void mostraVal() const;
              void incrementar();
}; // Fim de class Contador.
```

```
// Implementação.
// Construtor.
Contador::Contador() : vlrCont(0)
{
        cout << "\nConstruindo Contador...\n";
} // Fim de Contador::Contador()
// Destrutor.
Contador::~Contador()
{
        cout << "\nDestruindo Contador...\n";
} // Fim de Contador::~Contador()
unsigned int Contador::acessaVal() const
{
        return vlrCont;
} // Fim de Contador::acessaVal()
void Contador::defineVal(unsigned int val)
{
        vlrCont = val;
} // Fim de Contador::defineVal()
void Contador::mostraVal() const
{
        cout << "\nValor = "<< acessaVal();
} // Fim de Contador::mostraVal()
void Contador::incrementar()
{
        ++vlrCont;
} // Fim de Contador::incrementar()
int main()
{
// Um objeto contador.
        Contador umCont;
// Exibe valor.
        umCont.mostraVal();
// Incrementa.
        umCont.incrementar();
// Exibe novo valor.
        umCont.mostraVal();
        return 0;
} // Fim de main()
//----------------------------------------------------
```

Agora, acrescentaremos a nossa classe *Contador* o operador ++ em prefixo. Os operadores em prefixo podem ser sobrecarregados declarando-se uma função da forma:

tipoRetornado operator op(parametros);

Aqui, op é o operador a ser sobrecarregado. Assim, o operador ++ pode ser sobrecarregado com a seguinte sintaxe:

void operator++();

Isso é mostrado no exemplo abaixo.

```
//----------------------------------------------------
// Sobre03.cpp
// Ilustra sobrecarga de operadores.
// Acrescenta operador++
#include <iostream.h>
// Declara a classe Contador.
class Contador
{
```

```
        unsigned int vlrCont;
        public:
// Construtor.
              Contador();
// Destrutor.
              ~Contador();
              unsigned int acessaVal() const;
              void defineVal(unsigned int val);
              void mostraVal() const;
// Sobrecarrega operador.
              void operator++();
}; // Fim de class Contador.
// Implementação.
// Construtor.
Contador::Contador() :vlrCont(0)
{
        cout << "\nConstruindo Contador...\n";
} // Fim de Contador::Contador()
// Destrutor.
Contador::~Contador()
{
        cout << "\nDestruindo Contador...\n";
} // Fim de Contador::~Contador()
unsigned int Contador::acessaVal() const
{
        return vlrCont;
} // Fim de Contador::acessaVal()
void Contador::defineVal(unsigned int val)
{
        vlrCont = val;
} // Fim de Contador::defineVal()
void Contador::mostraVal() const
{
        cout << "\nValor = "<< acessaVal();
} // Fim de Contador::mostraVal()
void Contador::operator++()
{
        ++vlrCont;
} // Fim de Contador::operator++()
int main()
{
// Um objeto contador.
        Contador umCont;
// Exibe valor.
        umCont.mostraVal();
// Incrementa.
        ++umCont;
// Exibe novo valor.
        umCont.mostraVal();
        return 0;
} // Fim de main()
//--------------------------------------------------
```

O operador ++ em prefixo de nossa classe *Contador* está agora funcionando. Porém, ele tem uma séria limitação. Se quisermos colocar um *Contador* no lado direito de uma atribuição, isso não funcionará. Por exemplo:

Contador c = ++i;

O objetivo deste código é criar um novo *Contador*, c, e depois atribuir a ele o valor de i, depois de i ter sido incrementado. O construtor de cópia default cuidará da atribuição, mas atualmente, o operador de incremento não retorna um objeto *Contador*. Não podemos atribuir um objeto *void* a

um objeto *Contador*. É claro que o que precisamos é fazer com que o operador ++ retorne um objeto *Contador*.

```
//-----------------------------------------------------
// Sobre04.cpp
// Ilustra sobrecarga de operadores. Agora operador++
// retorna um objeto temporário.
#include <iostream.h>
// Declara a classe Contador.
class Contador
{
        unsigned int vlrCont;
        public:
// Construtor.
              Contador();
// Destrutor.
              ~Contador();
              unsigned int acessaVal() const;
              void defineVal(unsigned int val);
              void mostraVal() const;
              Contador operator++();
}; // Fim de class Contador.
// Implementação.
// Construtor.
Contador::Contador() : vlrCont(0)
{
        cout << "\nConstruindo Contador...\n";
} // Fim de Contador::Contador()
// Destrutor.
Contador::~Contador()
{
        cout << "\nDestruindo Contador...\n";
} // Fim de Contador::~Contador()
unsigned int Contador::acessaVal() const
{
        return vlrCont;
} // Fim de Contador::acessaVal()
void Contador::defineVal(unsigned int val)
{
        vlrCont = val;
} // Fim de Contador::defineVal()
void Contador::mostraVal() const
{
        cout << "\nValor = "<< acessaVal();
} // Fim de Contador::mostraVal()
Contador Contador::operator++()
{
        cout << "\nIncrementando...";
        ++vlrCont;
// Cria um objeto temporário.
        Contador temp;
        temp.defineVal(vlrCont);
        return temp;
} // Fim de Contador::operator++()
int main()
{
// Dois objetos Contador.
        Contador cont1, cont2;
// Exibe valor.
        cout << "\nObjeto cont1";
        cont1.mostraVal();
```

```
// Incrementa e atribui.
        cont2 = ++cont1;
// Exibe novo objeto.
        cout << "\nObjeto cont2";
        cont2.mostraVal();
        return 0;
} // Fim de main()
//----------------------------------------
```

Na verdade, não há necessidade do objeto *Contador* temporário criado no exemplo anterior. Se *Contador* tiver um construtor que recebe um valor, podemos simplesmente retornar o resultado desse construtor, como sendo o valor retornado pelo operador ++.
Vimos que o ponteiro *this* é passado para todas as funções membro de uma classe. Portanto, ele é passado também para o operador ++. Esse ponteiro aponta para o próprio objeto *Contador*, de modo que se for dereferenciado, ele retornará o objeto, já com o valor correto na variável *vlrCont*.
O ponteiro *this* pode então ser de-referenciado, evitando assim a necessidade de criação de um objeto temporário.

Sobrecarregando o operador =

O operador = é um dos operadores fornecidos por default pelo compilador, mesmo para os tipos (classes) definidos pelo programador. Mesmo assim, pode surgir a necessidade de sobrecarregá-lo.
Quando sobrecarregamos o operador =, precisamos levar em conta um aspecto adicional. Digamos que temos dois objetos Contador, c1 e c2. Com o operador de atribuição, podemos atribuir c2 a c1, da seguinte forma:

c1 = c2;

O que acontece se uma das variáveis membro for um ponteiro? E o que acontece com os valores originais de c1?
Lembremos o conceito de cópia rasa e cópia profunda. Uma cópia rasa apenas copia os membros, e os dois objetos acabam apontando para a mesma área do free store. Uma cópia profunda aloca a memória necessária.
Há ainda outra questão. O objeto c1 já existe na memória, e tem sua memória alocada. Essa memória precisa ser deletada, para evitar vazamentos de memória. Mas o que acontece se atribuirmos um objeto a si mesmo, da seguinte forma:

c1 = c1;

Ninguém vai fazer isso de propósito, mas se acontecer, o programa precisa ser capaz de lidar com isso. E mais importante, isso pode acontecer por acidente, quando referências e ponteiros de-referenciados ocultam o fato de que a atribuição está sendo feita ao próprio objeto.
Se essa questão não tiver sido tratada com cuidado, c1 poderá deletar sua memória alocada. Depois, no momento de copiar a memória do lado direito da atribuição, haverá um problema: a memória terá sido deletada. Para evitar esse problema, o operador de atribuição deve checar se o lado direito do operador de atribuição é o próprio objeto. Isso é feito examinando o ponteiro this.
O exemplo abaixo ilustra esse procedimento.

```
//--------------------------------------------------------
// SobreAtr.cpp
// Ilustra sobrecarga de operadores.
// Implementa operador =
#include <iostream.h>
// Declara a classe Contador.
class Contador
{
        unsigned int vlrCont;
        public:
```

```
// Construtor.
              Contador();
// Construtor com inicialização.
              Contador(unsigned int vlr);
// Destrutor.
              ~Contador();
              unsigned int acessaVal() const;
              void defineVal(unsigned int val);
              void mostraVal() const;
// O operador =
              Contador& operator=(const Contador&);
}; // Fim de class Contador.
// Implementação.
// Construtor.
Contador::Contador() :vlrCont(0)
{
} // Fim de Contador::Contador()
// Construtor com inicialização.
Contador::Contador(unsigned int vlr) :vlrCont(vlr)
{
} // Fim de Contador::Contador(unsigned int)
// Destrutor.
Contador::~Contador()
{
} // Fim de Contador::~Contador()
unsigned int Contador::acessaVal() const
{
        return vlrCont;
} // Fim de Contador::acessaVal()
void Contador::defineVal(unsigned int val)
{
        vlrCont = val;
} // Fim de Contador::defineVal()
void Contador::mostraVal() const
{
        cout << "\nValor = "<< acessaVal();
} // Fim de Contador::mostraVal()
// Operador =
Contador& Contador::operator=(const Contador& outro)
{
        vlrCont = outro.acessaVal();
        return *this;
} // Fim de Contador::operator=(const Contador&)
int main()
{
// Dois objetos Contador.
        Contador cont1(40), cont2(3);
// Exibe valores iniciais.
        cout << "\n*** Valores iniciais ***";
        cout << "\ncont1: ";
        cont1.mostraVal();
        cout << "\ncont2: ";
        cont2.mostraVal();
// Atribui.
        cont1 = cont2;
// Exibe novos valores.
        cout << "\n*** Apos atribuicao ***";
        cout << "\ncont1: ";
        cont1.mostraVal();
        cout << "\ncont2: ";
        cont2.mostraVal();
```

```
        return 0;
} // Fim de main()
//--------------------------------------------------
```

Conversão entre objetos e tipos simples

O que acontece quando tentamos converter uma variável de um tipo simples, como int, em um objeto de uma classe definida pelo programador?
A classe para a qual desejamos converter o tipo simples precisará ter um construtor especial, com essa finalidade. Esse construtor deverá receber como argumento o tipo simples a ser convertido.
Como é feita a conversão em sentido oposto, de um objeto para um tipo simples?
Para solucionar esse tipo de problema, C++ oferece a possibilidade de acrescentar a uma classe os operadores de conversão. Isso permite que uma classe especifique como devem ser feitas conversões implícitas para tipos simples. O exemplo abaixo ilustra a situação.

```
//--------------------------------------------------------
// ConvObj.cpp
// Ilustra conversão de um tipo simples em um objeto.
#include <iostream.h>
// Declara a classe Contador.
class Contador
{
        unsigned int vlrCont;
        public:
// Construtor.
        Contador();
// Destrutor.
        ~Contador();
        unsigned int acessaVal() const;
        void defineVal(unsigned int val);
        void mostraVal() const;
}; // Fim de class Contador.
// Implementação.
// Construtor.
Contador::Contador() :vlrCont(0)
{
} // Fim de Contador::Contador()
// Destrutor.
Contador::~Contador()
{
} // Fim de Contador::~Contador()
unsigned int Contador::acessaVal() const
{
        return vlrCont;
} // Fim de Contador::acessaVal()
void Contador::defineVal(unsigned int val)
{
        vlrCont = val;
} // Fim de Contador::defineVal()
void Contador::mostraVal() const
{
        cout << "\nValor = "<< acessaVal();
} // Fim de Contador::mostraVal()
int main()
{
// Uma variável unsigned int.
        unsigned int uiVar = 50;
// Um objeto Contador.
        Contador cont;
```

```
// Tenta converter unsigned int em contador.
        cont.defineVal(uivar);
// Exibe valor.
        cout << "\nValor de cont: ";
        cont.mostraVal();
        return 0;
} // Fim de main()
//--------------------------------------------------
```

Arrays de objetos

Qualquer objeto, seja de um tipo simples ou de uma classe definida pelo programador, pode ser armazenado em um array. Quando declaramos o array, informamos ao compilador qual o tipo de objeto a ser armazenado, bem como o tamanho do array. O compilador sabe então quanto espaço alocar, dependendo do tipo de objeto. No caso de objetos de uma classe, o compilador sabe quanto espaço precisa ser alocado para cada objeto com base na declaração da classe. A classe deve ter um construtor default, que não recebe nenhum argumento, de modo que os objetos possam ser criados quando o array é definido.

O acesso aos membros de dados em um array de objetos é um processo em dois passos. Primeiro, identificamos o membro do array, com o auxílio do operador de índice [], e depois aplicamos o operador ponto . para acessar o membro.

```
Exemplo
//----------------------------------------------------------
// ArrObj.cpp
// Ilustra o uso de arrays de objetos.
#include <iostream.h>
class Cliente
{
        int numCliente;
        float saldo;
        public:
// Construtor.
              Cliente();
              int acessaNum() const;
              float acessaSaldo() const;
              void defineNum(int num);
              void defineSaldo(float sal);
}; // Fim de class Cliente.
// Definições.
Cliente::Cliente()
{
        numCliente = 0;
        saldo = 0.0;
} // Fim de Cliente::Cliente()
int Cliente::acessaNum() const
{
        return numCliente;
} // Fim de Cliente::acessaNum()
float Cliente::acessaSaldo() const
{
        return saldo;
} // Fim de Cliente::acessaSaldo()
void Cliente::defineNum(int num)
{
        numCliente = num;
} // Fim de Cliente::defineNum()
void Cliente::defineSaldo(float sal)
{
```

```
        saldo = sal;
} // Fim de Cliente::defineSaldo()
int main()
{
// Um array de clientes.
        Cliente arrayClientes[5];
// Inicializa.
        for(int i = 0; i < 5; i++)
        {
              arrayClientes[i].defineNum(i + 1);
              arrayClientes[i].defineSaldo((float)(i + 1) * 25);
        } // Fim de for(int i...
// Exibe.
        for(int i = 0; i < 5; i++)
        {
              cout << "\nCliente: "<< arrayClientes[i].acessaNum()<<
      "\tSaldo = "<< arrayClientes[i].acessaSaldo();
      } // Fim de for(int i = 1...
      return 0;
} // Fim de main()
//-------------------------------------------------------
```

Uma classe string

Atualmente, todos os compiladores C++ em conformidade com o padrão ANSI/ISO vêm com uma classe string, o que facilita bastante a manipulação de strings. Entretanto, como exercício de programação vamos implementar nossa própria classe string. Para evitar confusões, vamos chamá-la de ClString.

```
Exemplo
//------------------------------------------------------------
// ClStr.cpp
// Ilustra um exemplo de classe string.
#include <iostream.h>
#include <string.h>
// Declara a classe.
class ClString
{
// A string propriamente dita.
        char* str;
// O comprimento da string.
        unsigned int compr;
// Um construtor private.
        ClString(unsigned int);
        public:
// Construtores.
              ClString();
              ClString(const char* const);
              ClString(const ClString&);
// Destrutor.
              ~ClString();
// Operadores sobrecarregados.
              Char& operator[](unsigned int posicao);
              char operator[](unsigned int posicao) const;
              ClString operator+(const ClString&);
              void operator +=(const ClString&);
              ClString& operator =(const ClString&);
// Métodos de acesso.
              unsigned int acessaCompr() const
              {
```

```
                        return compr;
                } // Fim de acessaCompr()
                const char* acessaStr() const
                {
                        return str;
                } // Fim de acessaStr()
}; // Fim de class ClString.
// Implementações.
// Construtor default.
// Cria uma string de comprimento zero.
ClString::ClString()
{
// cout << "\nConstrutor default...\n";
        str = new char[1];
        str[0] = '\0';
        compr = 0;
} // Fim de ClString::ClString()
// Construtor private.
// Usado somente pelos métodos da classe.
// Cria uma string com o comprimento especificado
// e a preenche com o caractere '\0'
ClString::ClString(unsigned int comp)
{
// cout << "\nConstrutor private...\n";
        str = new char[comp + 1];
        for(unsigned int i = 0; i <= comp; i++)
        str[i] = '\0';
        compr = comp;
} // Fim de ClString::ClString(unsigned int)
// Constroi um objeto ClString
// a partir de um array de caracteres.
ClString::ClString(const char* const cArray)
{
// cout << "\nConstruindo de array...\n";
        compr = strlen(cArray);
        str = new char[compr + 1];
        for(unsigned int i = 0; i < compr; i++) str[i] = cArray[i];
        str[compr] = '\0';
} // Fim de ClString::ClString(const char* const)
// Construtor de cópia.
ClString::ClString(const ClString& strRef)
{
// cout << "\nConstrutor de copia...\n";
        compr = strRef.acessaCompr();
        str = new char[compr + 1];
        for(unsigned int i = 0; i < compr; i++) str[i] = strRef[i];
        str[compr] = '\0';
} // Fim de ClString::ClString(const String&)
// Destrutor.
ClString::~ClString()
{
// cout << "\nDestruindo string...\n";
        delete[] str;
        compr = 0;
} // Fim de ClString::~ClString()
// Operador =
// Libera memória atual; Copia string e compr.
ClString& ClString::operator=(const ClString& strRef)
{
        if(this == &strRef)
        return *this;
```

```
        delete[] str;
        compr = strRef.acessaCompr();
        str = new char[compr + 1];
        for(unsigned int i = 0; i < compr; i++)
        str[i] = strRef[i];
        str[compr] = '\0';
        return *this;
} // Fim de ClString::operator=(const ClString&)
// Operador de posição não-constante.
// Referência permite que char seja modificado.
char& ClString::operator[](unsigned int pos)
{
        if(pos > compr)
              return str[compr - 1];
        else
              return str[pos];
} // Fim de ClString::operator[]()
// Operador de posição constante.
// Para uso em objetos const.
char ClString::operator[](unsigned int pos) const
{
        if(pos > compr)
              return str[compr - 1];
        else
              return str[pos];
} // Fim de ClString::operator[]()
// Concatena duas strings.
ClString ClString::operator+(const ClString& strRef)
{
        unsigned int comprTotal = compr + strRef.acessaCompr();
        ClString tempStr(comprTotal);
        unsigned int i;
        for(i = 0; i < compr; i++)tempStr[i] = str[i];
        for(unsigned int j = 0;j < strRef.acessaCompr();j++, i++)
              tempStr[i] = strRef[j];
        tempStr[comprTotal] = '\0';
        return tempStr;
} // Fim de ClString::operator+()
void ClString::operator+=(const ClString& strRef)
{
        unsigned int comprRef = strRef.acessaCompr();
        unsigned int comprTotal = compr + comprRef;
        ClString tempStr(comprTotal);
        unsigned int i;
        for(i = 0; i < compr; i++)tempStr[i] = str[i];
        for(unsigned int j = 0;j < strRef.acessaCompr();j++, i++)
              tempStr[i] = strRef[i - compr];
        tempStr[comprTotal] = '\0';
        *this = tempStr;
} // Fim de ClString::operator+=()
int main()
{
// Constroi string a partir de array de char.
        ClString str1("POO na UAb");
// Exibe.
        cout << "str1:\t" << str1.acessaStr() << '\n';
// Atribui array de chars a objeto ClString.
        char* arrChar = "O rato roeu ";
        str1 = arrChar;
// Exibe.
        cout << "str1:\t" << str1.acessaStr() << '\n';
```

```
// Cria um segundo array de chars.
        char arrChar2[64];
        strcpy(arrChar2, "a roupa do rei.");
// Concatena array de char com objeto ClString.
        str1 += arrChar2;
// Exibe.
        cout << "arrChar2:\t" << arrChar2 << '\n';
        cout << "str1:\t" << str1.acessaStr() << '\n';
// Coloca R maiúsculo.
        str1[2] = str1[7] = str1[14] = str1[23] = 'R';
// Exibe.
        cout << "str1:\t" << str1.acessaStr() << '\n';
        return 0;
} // Fim de main()
//-----------------------------------------------------
```

Exemplo de lista encadeada

Os arrays são muito práticos, mas têm uma séria limitação: seu tamanho é fixo. Ao definir o tamanho do array, se errarmos pelo excesso, estaremos desperdiçando espaço de armazenamento. Se errarmos pela falta, o conteúdo pode estourar o tamanho do array, criando sérios problemas.
Uma forma de contornar essa limitação é com o uso de uma lista encadeada. Uma lista encadeada é uma estrutura de dados que consiste de pequenos containers, que podem ser encadeados conforme necessário. A idéia da lista encadeada é que ela pode conter um objeto de uma determinada classe. Se houver necessidade, podemos acrescentar mais objetos, fazendo com que o último objeto da lista aponte para o novo objeto, recém acrescentado. Ou seja, criamos um container para cada objeto e encadeamos esses containers conforme a necessidade.
Os containers são chamados de nós. O primeiro nó é chamado cabeça da lista; o último nó é chamado de cauda.
As listas podem ser de três tipos:
(a) Simplesmente encadeadas
(b) Duplamente encadeadas
(c) Árvores
Em uma lista simplesmente encadeada, cada nó aponta para o nó seguinte, mas o nó seguinte não aponta para o nó anterior. Para encontrar um determinado nó, começamos da cabeça da lista e seguimos nó por nó. Uma lista duplamente encadeada permite movimentar-se para a frente e para trás na cadeia. Uma árvore é uma estrutura complexa, construída com nós, sendo que cada um deles aponta para dois ou três outros nós. O exemplo abaixo ilustra a construção de uma lista simplesmente encadeada.

```
//--------------------------------------------------
// ListEnc.cpp
// Ilustra a criação de uma lista encadeada.
#include <iostream.h>
#include <assert.h>
// Define a classe de  objetos que formarão a lista.
class Cliente
{
        int numCliente;
        public:
// Construtores.
        Cliente() {numCliente = 1;}
        Cliente(int num) : numCliente(num){}
// Destrutor.
        ~Cliente(){}
// Método de acesso.
        int acessaNum() const {return numCliente;}
```

```
}; // Fim de class Cliente.
// Uma classe para gerir e ordenar a lista.
class No
{
        Cliente* pCliente;
        No* proxNo;
        public:
// Construtor.
              No(Cliente*);
// Destrutor.
              ~No();
// Outros métodos.
        void defineProx(No* pNo) {proxNo = pNo;}
        No* acessaProx() const {return proxNo;}
        Cliente* acessaCli() const {return pCliente;}
        void insere(No*);
        void Exibe();
}; // Fim de class No.
// Implementação.
// Construtor.
No::No(Cliente* pCli): pCliente(pCli), proxNo(0)
{
} // Fim de No::No(Cliente*)
// Destrutor.
No::~No()
{
        cout << "Deletando No...\n";
        delete pCliente;
        pCliente = 0;
        delete proxNo;
        proxNo = 0;
} // Fim de No::~No()
// Método insere()
// Ordena clientes pelo número.
// Algoritmo: Se este cliente é o último da fila,
// acrescenta o novo cliente. Caso contrário,
// se o novo cliente tem número maior que o
// cliente atual e menor que o próximo da fila,
// insere-o depois deste. Caso contrário,
// chama insere() para o próximo cliente da fila.
void No::insere(No* novoNo)
{
        if(!proxNo)
              proxNo = novoNo;
        else
        {
              int numProxCli = proxNo->acessaCli()->acessaNum();
              int novoNum = novoNo->acessaCli()->acessaNum();
              int numDeste = pCliente->acessaNum();
              assert(novoNum >= numDeste);
              if(novoNum < numProxCli)
              {
                    novoNo->defineProx(proxNo);
                    proxNo = novoNo;
              } // Fim de if(novoNum < numProxCli)
              else
                    proxNo->insere(novoNo);
        } // Fim de else (externo)
} // Fim de No::insere(No*)
void No::Exibe()
{
```

```
        if(pCliente->acessaNum() > 0)
        {
              cout << "Num. do Cliente = ";
              cout << pCliente->acessaNum() << "\n";
        } // Fim de if(pCliente->...
        if(proxNo)proxNo->Exibe();
} // Fim de No::Exibe()
int main()
{
// Um ponteiro para nó.
        No* pNo;
// Um ponteiro para Cliente, inicializado.
        Cliente* pontCli = new Cliente(0);
// Um array de ints para fornecer números de clientes.
        int numeros[] = {19, 48, 13, 17, 999, 18, 7, 0};
// Exibe números.
        cout << "\n*** Nums. fornecidos ***\n";
        for(int i = 0; numeros[i]; i++) cout << numeros[i] << " ";
        cout << "\n";
        No* pCabeca = new No(pontCli);
// Cria alguns nós.
        for(int i = 0; numeros[i] != 0; i++)
        {
              pontCli = new Cliente(numeros[i]);
              pNo = new No(pontCli);
              pCabeca->insere(pNo);
        } // Fim de for(int i = 0...
        cout << "\n*** Lista ordenada ***\n";
        pCabeca->Exibe();
        cout << "\n*** Destruindo lista ***\n";
        delete pCabeca;
        cout << "\n*** Encerrando... ***\n";
        return 0;
} // Fim de main()
//----------------------------------------------------
```

Programação Orientada por Objetos – Tópico 5

***Temática**: Herança simples*

***Actividade 5**: Familiarização com o conceito de herança simples*

Competências a desenvolver:

- Compreender o conceito de herança
- Aprender a criar classes e métodos com o mecanismo de herança

Herança

Provavelmente herança é o recurso que torna o conceito de classe mais poderoso. Em C++, o termo herança se aplica apenas às classes. Variáveis não podem herdar de outras variáveis e funções não podem herdar de outras funções.
Herança permite que se construa e estenda continuamente classes desenvolvidas por você mesmo ou por outras pessoas, sem nenhum limite. Começando da classe mais simples, pode-se derivar classes cada vez mais complexas que não são apenas mais fáceis de debuggar, mas elas próprias são mais simples.
O objetivo de um projeto em C++ é desenvolver classes que resolvam um determinado problema.Estas classes são geralmente construídas incrementalmente começando de uma classe básica simples, através de herança. Cada vez que se deriva uma nova classe começando de uma já existente, pode-se herdar algumas ou todas as características da classe pai, adicionando novas quando for necessário. Um projeto completo pode ter centenas de classes, mas normalmente estas classes são derivadas de algumas poucas classes básicas. C++ permite não apenas herança simples, mas também múltipla, permitindo que uma classe incorpore comportamentos de todas as suas classes bases.
Reutilização em C++ se dá através do uso de uma classe já existente ou da construção de uma nova classe a partir de uma já existente.

Classes derivadas

A descrição anterior pode ser interessante, mas um exemplo é a melhor forma de mostrar o que é herança e como ela funciona. Aqui está um exemplo de duas classes, a segunda herdando as propriedades da primeira:

```
class Caixa {
        public:
              int altura, largura;
              void Altura(int a) { altura=a; }
              void Largura(int l) { largura=l; }
};
class CaixaColorida : public Caixa {
        public:
              int cor;
              void Cor(int c) { cor=c; }
};
```

Usando a terminologia de C++, a classe *Caixa* é chamada classe base para a classe *CaixaColorida*, que é chamada classe derivada. Classes base são também chamadas de classes pai. A classe *CaixaColorida* foi declarada com apenas uma função, mas ela herda duas funções e duas variáveis da classe base. Sendo assim, o seguinte código é possível:

```
void main()
{
        CaixaColorida cc;
        cc.Cor(5);
```

```
        cc.Largura(3); // herdada
        cc.Altura(50); // herdada
}
```

Note que as funções herdadas são udadas exatamente como as não herdadas. A classe Colorida não precisou sequer mencionar o fato de que as funções Caixa::Altura() e Caixa::Largura() foram herdadas. Esta uniformidade de expressão é um grande recurso de C++. Usar um recurso de uma classe não requer que se saiba se este recurso foi herdado ou não, já que a notação é invariante. Em muitas classes pode existir uma cadeia de classes base derivadas de outras classes base. Uma classe herdada de uma árvore de herança como esta herdaria características de muitas classes pai diferentes. Entretanto, em C++, não é preciso se preocupar onde ou quando um recurso foi introduzido na árvore.

Derivar uma classe de outra aumenta a flexibilidade a um custo baixo. Uma vez que já existe uma classe base sólida, apenas as mudanças feitas nas classes derivadas precisam ser depuradas. Mas quando exatamente se usa uma classe base, e que tipos de modificações precisam ser feitas? Quando se herda característias de uma classe base, a classe derivada pode estender, restringir, modificar, eliminar ou usar qualquer dos recursos sem qualquer modificação.

O que não é herdado

Nem tudo é herdado quando se declara uma classe derivada. Alguns casos são inconsistentes com herança por definição:

- Construtores
- Destrutores
- Operadores new
- Operadores de atribuição (=)
- Relacionamentos *friend*
- Atributos privados

Classes derivadas invocam o construtor da classe base automaticamente, assim que são instanciadas.

Membros de classes protected

Na seção de controle de acesso, vimos como deixar disponíveis ou ocultar atributos das classes, usando os especificadores *public* e *private*. Além desses dois, existe um outro especificador, *protected*. Do ponto de vista de fora da classe, um atributo *protected* funciona como *private*: não é acessível fora da classe; a diferença está na herança. Enquanto um atributo *private* de uma classe base não é visível na classe derivada, um *protected* é, e continua sendo *protected* na classe derivada. Por exemplo:

```
class A {
        private:
                int a;
      protected:
                int b;
        public:
                int c;
};
class B : public A {
        public:
                int geta() { return a; } // ERRO!! a não é visível
                int getb() { return b; } // válido (b protected)
                int getc() { return c; } // válido (c public)
};
void main()
{
```

```
        A ca;
        B cb;
        ca.a = 1; // ERRO! a não é visível (private)
        ca.b = 2; // ERRO! b não é visível de fora (protected)
        ca.c = 3; // válido (c é public)
        cb.a = 4; // ERRO! a não é visível nem internamente em B
        cb.b = 5; // ERRO! b continua protected em B
        cb.c = 6; // válido (c continua public em B)
}
```

Construtores e destrutores

Quando uma classe é instanciada, seu construtor é chamado. Se a classe foi derivada de alguma outra, o construtor da classe base também precisa ser chamado. A ordem de chamada dos construtores é fixa em C++. Primeiro a classe base é construída, para depois a derivada ser construída. Se a classe base também deriva de alguma outra, o processo se repete recursivamente até que uma classe não derivada é alcançada.
Desta forma, quando um construtor para uma classe derivada é chamado, todos os procedimentos efetuados pelo construtor da classe base já foram realizados. Considere a seguinte árvore de herança:

class Primeira {};
class Segunda: public Primeira {};
class Terceira: public Segunda {};

Quando a classe Terceira é instanciada, os construtores são chamados da seguinte maneira:

Primeira::Primeira();
Segunda::Segunda();
Terceira::Terceira();

Esta ordem faz sentido, já que uma classe derivada é uma especialização de uma classe mais genérica. Isto significa que o construtor de uma classe derivada pode usar atributos herdados.
Os destrutores são chamados na ordem inversa dos construtores. Primeiro, os atributos mais especializados são destruídos, depois os mais gerais. Então a ordem de chamada dos destrutores quando Terceira sai do âmbito é:

Terceira::~Terceira();
Segunda::~Segunda();
Primeira::~Primeira();

Como os construtores das classes base são chamados automaticamente, deve existir alguma maneira de passar os argumentos corretos para estes construtores, no caso de eles necessitarem de parâmetros. Existe uma notação especial para este caso, ilustrada abaixo, para funções *inline* e não *inline*:

```
class Primeira {
        int a, b, c;
        public:
              Primeira(int x, int y, int z) { a=x; b=y; c=z; }
};
class Segunda : public Primeira {
        int valor;
        public:
              Segunda(int d) : Primeira(d, d+1, d+2) { valor = d; }
              Segunda(int d, int e);
```

```
};
Segunda::Segunda(int d, int e) : Primeira(d, e, 13)
{
        valor = d + e;
}
```
A partir do exemplo acima, não é difícil perceber que, se uma classe base não possui um construtor sem parâmetros, a classe derivada tem que, obrigatoriamente, declarar um construtor, mesmo que este construtor seja vazio:

```
class Base {
        protected:
              int valor;
        public:
              Base(int a) { valor = a; }
// esta classe não possui um construtor sem parâmetros
};
class DerivadaErrada : public Base{
        public:
              int pegaValor() { return valor; }
// ERRO! classe não declarou construtor, compilador não
// sabe que parâmetro passar para Base
};
class DerivadaCerta: public Base {
        public:
              int pegaValor() { return valor; }
              DerivadaCerta() : Base(0) {}
// CERTO: mesmo que não haja nada a fazer para inicializar a classe,
// é necessário declarar um construtor para dizer com que parâmetro
// construir a classe Base
};
```

Herança pública x herança privada

Nos exemplos acima, em toda declaração de uma classe derivada, usou-se a palavra *public*:
class B : public A { ...
Na realidade, os especificadores de acesso *private* e *public* podem ser usados na declaração de uma herança. Por default, as heranças são *private*, por isso usou-se *public* nos exemplos acima.
Estes especificadores afetam o nível de acesso que os atributos terão na classe derivada:

- *private*: todos os atributos herdados (*public*, *protected*) tornam-se *private* na classe derivada;
- *public*: todos os atributos *public* são *public* na classe derivada, e todos os *protected* também continuam *protected*.

Na realidade, isto é uma consequência da finalidade real de heranças *public* e *protected*, que voltará a ser discutida em compatibilidade de tipos. O exemplo abaixo ilustra de forma mais detalhada a criação da herança entre classes.

```
//---------------------------------------------------------
// IntrHer.cpp
// Apresenta o uso de herança.
#include <iostream.h>
class ClasseBase
{
        protected:
              int m_propr_base1;
              int m_propr_base2;
        public:
// Construtores.
              ClasseBase() : m_propr_base1(10),m_propr_base2(20) {}
// Destrutor.
```

```
                ~ClasseBase() {}
// Métodos de acesso.
                int acessaPropr1() const {return m_propr_base1;}
                void definePropr1(int valor){ m_propr_base1 = valor;}
                int acessaPropr2() const {return m_propr_base2;}
                void definePropr2(int valor){m_propr_base2 = valor;}
// Outros métodos.
                void met_base1() const
                {
                      cout << "\nEstamos em met_base1...\n";
                } // Fim de met_base1()
                void met_base2() const
                {
                      cout << "\nEstamos em met_base2...\n";
                } // Fim de met_base2()
}; // Fim de class ClasseBase
class ClasseDeriv : public ClasseBase
{
        private:
                int m_propr_deriv;
        public:
// Construtor.
                ClasseDeriv() : m_propr_deriv(1000){}
// Destrutor.
                ~ClasseDeriv() {};
// Métodos de acesso.
                int acessaPropr_deriv() const
                {
                      return m_propr_deriv;
                } // Fim de acessaPropr_deriv()
                void definePropr_deriv(int valor)
                {
                      m_propr_deriv = valor;
                } // Fim de definePropr_deriv()
// Outros métodos.
                void metodoDeriv1()
                {
                      cout << "Estamos em metodoDeriv1()...\n";
                } // Fim de metodoDeriv1()
                void metodoDeriv2()
                {
                      cout << "Estamos em metodoDeriv2()...\n";
                } // Fim de metodoDeriv2()
}; // Fim de class ClasseDeriv.
int main()
{
// Cria um objeto de ClasseDeriv.
        ClasseDeriv objDeriv;
// Chama métodos da classe base.
        objDeriv.met_base1();
        objDeriv.met_base2();
// Chama métodos da classe derivada.
        objDeriv.metodoDeriv1();
        cout << "Valor de m_propr_deriv = "<< objDeriv.acessaPropr_deriv()
<< "\n";
        return 0;
} // Fim de main()
//-----------------------------------------------------
```

Argumentos para construtores da classe base

Muitas vezes, ao criar um objeto, utilizamos um construtor que recebe parâmetros. Já vimos que, quando se trata de uma classe derivada, o construtor da classe base sempre é chamado. E se o construtor da classe base precisar receber parâmetros?
Como faremos para passar os parâmetros certos para o construtor da classe base?
O exemplo abaixo ilustra como isso é feito.

```
//--------------------------------------------------------------------------
// ArgCstr.cpp
// Ilustra a passagem de args para construtores da classe base.
#include <iostream.h>
enum VALORES {VLR1, VLR2, VLR3, VLR4, VLR5, VLR6};
class ClasseBase
{
        protected:
              int m_propr_base1;
              int m_propr_base2;
        public:
// Construtores.
              ClasseBase();
              ClasseBase(int valor);
// Destrutor.
              ~ClasseBase();
// Métodos de acesso.
              int acessaPropr1() const {return m_propr_base1;}
              void definePropr1(int valor){ m_propr_base1 = valor;}
              int acessaPropr2() const {return m_propr_base2;}
              void definePropr2(int valor){m_propr_base2 = valor;}
// Outros métodos.
              void met_base1() const
              {
                    cout << "\nEstamos em met_base1...\n";
              } // Fim de met_base1()
              void met_base2() const
              {
                    cout << "\nEstamos em met_base2...\n";
              } // Fim de met_base2()
}; // Fim de class ClasseBase
class ClasseDeriv : public ClasseBase
{
        private:
              VALORES m_propr_deriv;
        public:
// Construtores.
              ClasseDeriv();
              ClasseDeriv(int propBase1);
              ClasseDeriv(int propBase1, int propBase2);
// Destrutor.
              ~ClasseDeriv();
// Métodos de acesso.
              VALORES acessaPropr_deriv() const
              {
                    return m_propr_deriv;
              } // Fim de acessaPropr_deriv()
              void definePropr_deriv(VALORES valor)
              {
                    m_propr_deriv = valor;
              } // Fim de definePropr_deriv()
// Outros métodos.
              void metodoDeriv1()
```

```
		{
			cout << "Estamos em metodoDeriv1()...\n";
		} // Fim de metodoDeriv1()
		void metodoDeriv2()
		{
			cout << "Estamos em metodoDeriv2()...\n";
		} // Fim de metodoDeriv2()
}; // Fim de class ClasseDeriv.
// Implementações.
ClasseBase::ClasseBase() : m_propr_base1(10),m_propr_base2(20)
{
	cout << "\nConstrutor ClasseBase()...\n";
} // Fim de ClasseBase::ClasseBase()
ClasseBase::ClasseBase(int propr1) : m_propr_base1(propr1),
m_propr_base2(20)
{
	cout << "\nConstrutor ClasseBase(int)...\n";
} // Fim de ClasseBase::ClasseBase(int)
// Destrutor.
ClasseBase::~ClasseBase()
{
	cout << "\nDestrutor ~ClasseBase()...\n";
} // Fim de ClasseBase::~ClasseBase()
// Construtores ClasseDeriv()
ClasseDeriv::ClasseDeriv() :ClasseBase(), m_propr_deriv(VLR3)
{
	cout << "\nConstrutor ClasseDeriv()\n";
} // Fim de ClasseDeriv::ClasseDeriv()
ClasseDeriv::ClasseDeriv(int propBase1) :ClasseBase(propBase1),
m_propr_deriv(VLR3)
{
	cout << "\nConstrutor ClasseDeriv(int)\n";
} // Fim de ClasseDeriv::ClasseDeriv(int)
ClasseDeriv::ClasseDeriv(int propBase1, int propBase2) :
ClasseBase(propBase1), m_propr_deriv(VLR3)
{
	m_propr_base2 = propBase2;
	cout << "\nConstrutor ClasseDeriv(int, int)\n";
} // Fim de ClasseDeriv::ClasseDeriv(int, int)
// Destrutor.
ClasseDeriv::~ClasseDeriv()
{
	cout << "\nDestrutor ~ClasseDeriv()\n";
} // Fim de ClasseDeriv::~ClasseDeriv()
int main()
{
// Cria 3 objetos de ClasseDeriv.
	ClasseDeriv objDeriv1;
	ClasseDeriv objDeriv2(2);
	ClasseDeriv objDeriv3(4, 6);
// Chama métodos da classe base.
	objDeriv1.met_base1();
	objDeriv2.met_base2();
// Exibe valores.
	cout << "\nValores de objDeriv3: "<< objDeriv3.acessaPropr1()
<< ", "<< objDeriv3.acessaPropr2()<< ", "<< objDeriv3.acessaPropr_deriv();
	return 0;
} // Fim de main()
//--------------------------------------------------
```

AbERTA
www.uab.pt

Programação Orientada por Objetos – Tópico 6

***Temática**: Polimorfismo e herança composta*

Actividade 6: Familiarização com o conceito de polimorfismo e herança composta

Competências a desenvolver:

- Compreender o conceito de herança múltipla e polimorfismo
- Aprender a criar classes com o mecanismo de herança múltipla
- Aprender a criar classes virtuais

Polimorfismo

Polimorfismo descreve a capacidade de um código C++ se comportar de diferentes formas dependendo do contexto em tempo de execução. Este é um dos recursos mais poderosos de linguagens orientadas a objetos (se não o mais), que permite trabalhar em um nível de abstração bem alto ao mesmo tempo que facilita a incorporação de novos pedaços em um sistema já existente. Em C++ o polimorfismo se dá através da conversão de ponteiros (ou referências) para objetos.

Conversão de ponteiros

Normalmente se usam não objetos de classes isoladas, mas sim objetos em uma hierarquia de classes.

Considere as seguintes classes:

```
class A {
        public: void f();
};

class B: public A {
        public: void g();
};
```

Como B é derivado de A, todos os membros disponíveis em A (função f) também estarão disponíveis em B. Então B é um superconjunto de A, e todas as operações que podem ser feitas com objetos da classe A também podem ser feitas com objetos do tipo B. A classe B é uma especialização da classe A, e é não só um objeto do tipo B, mas também um objeto do tipo A. Nada impede que objetos da classe B sejam vistos como sendo da classe A, pois todas as operações válidas para A são também válidas para B.
Ver um objeto do tipo B como sendo do tipo A significa convertê-lo para o tipo A. Esta conversão pode ser feita, sempre no sentido da classe mais especializada para a mais básica. A conversão inversa não é permitida, pois operações específicas de B não são válidas sobre objetos da classe A. Conversão aqui não deve ser entendida como cópia. A simples atribuição de um objeto do tipo B para um objeto do tipo A copia a parte A do objeto do tipo B para o objeto do tipo A. O polimorfismo é feito através da conversão de ponteiros.

O exemplo abaixo mostra as várias alternativas:

```
void main()
{
        A a, *pa; // pa pode apontar para objetos do tipo A e derivados
        B b, *pb; // pb pode apontar para objetos do tipo B e derivados
        a = b; // copia a parte A de b para a (não é conversão)
        b = a; // erro! a pode não ter todos elementos para a cópia
        pa = &a; // ok
        pa = &b; // ok, pa aponta para um objeto do tipo B
```

```
        pb = pa; // erro! pa pode apontar para um objeto do tipo A
        pb = &b; // ok
        pb = &a; // erro! pb não pode apontar para objetos do tipo A
}
```

Para tirar qualquer dúvida sobre quais conversões podem ser feitas, o exemplo abaixo mostra o que pode ser feito com este recurso a partir das classes A e B:

```
void chamaf(A* a) // pode ser chamada para A e derivados
{
        a->f();
}
void chamag(B* b) // pode ser chamada para B e derivados
{
        b->g();
}
void main()
{
        A a;
        B b;
        chamaf(&a); // ok, a tem a função f
        chamag(&a); // erro! a não tem a função g
// (a não pode ser convertido para o tipo B)
        chamaf(&b); // ok, b tem a função f
        chamag(&b); // ok, b tem a função g
}
```

Repare que as funções chamaf e chamag foram escritas para os tipos A e B, mas podem ser usadas com qualquer objeto que seja derivado destes. Se um novo objeto derivado de B for criado no futuro, a mesma função poderá ser usada sem necessidade de recompilação. Estas conversões só podem ser feitas quando a herança é pública. Se a herança for privada a conversão não é permitida. Redefinição de métodos em uma hierarquia Não existe sobrecarga em uma hierarquia. A definição de um método com mesmo nome de uma classe básica não deixa os dois disponíveis, mesmo que os tipos dos parâmetros sejam diferentes. Os métodos da classe básica de mesmo nome são escondidos. Eles não ficam inacessíveis, mas não podem ser chamados diretamente:

```
class A {
        public: void f(){};
};
class B : public A {
        public:
              void f(int a){}; // f(int) esconde f()
              void f(char* str){};
};
void main()
{
        B b;
        b.f(10); // ok, função f(int) de B
        b.f("abc"); // ok, função f(char*) de B
        b.f(); // erro! f(int) escondeu f()
        b.A::f(); // ok
}
```

É possível também declarar um método com mesmos nome e assinatura (tipo de retorno e tipo dos parâmetros) que um da classe base. O novo método esconde o da classe base, que precisa do operador de escopo para ser acessado. No entanto, esta redefinição merece atenção especial. Considerando o exemplo:

```
class A {
```

```
        public: void f(){};
};
class B : public A {
        public: void f(){};
};
void chamaf(A* a) { a->f(); }
void main()
{
        B b;
        chamaf(&b);
}
```

A função chamaf pode ser usada para qualquer objeto do tipo A e derivados. No exemplo acima, ela é chamada com um objeto do tipo B. O método f é chamado no corpo de chamaf. Mas qual versão será executada? No exemplo acima o método executado será A::f.
Isto é o que acontece, mas será que este é o comportamento desejado? Se o polimorfismo nesse caso for encarado como uma maneira diferente (mais limitada) de ver o mesmo objeto, não seria natural chamar o método B::f? Afinal, o objeto é do tipo B, apenas está "guardado" em um ponteiro para o tipo A.

Polimorfismo, override e overload

Nesse ponto, é importante entender bem como estes conceitos estão interligados:

- O conceito de polimorfismo indica as várias formas que vários objetos se comportam mesmo sendo todos eles oriundo de uma mesma classe, resumindo uma única classe pode dar origem a vários objetos cujos os comportamentos (métodos) podem ter implementações diferentes.
- Overload, é uma palavra inglesa que em português significa sobrecarga. A sobrecarga de métodos consiste em se declarar vários métodos em uma única classe ou em classes descendentes, com o mesmo nome, porém eles tem que ter assinaturas diferentes. A assinatura do método é o âmbito do método identificado pelo seu retorno, pelo seu nome, bem como o recebimento ou não de seus parâmetros. Um exemplo:

```
public class Teste {

     public void fazAlgo() {
           System.out.println("Este método não recebe parâmetro);
     }

     public void fazAlgo(String mensagem) {
           System.out.println("Mensagem");
     }

}
```

- Override, é uma palavra inglesa que significa "Sobrescrita" em português. A sobrescrita **somente é possível em classes herdadas**, isto porque este conceito significa o que o próprio nome já diz, ou seja, substituir um método da super classe na sub classe sobrescrevendo-o (substituindo-o) o mesmo. Vamos ao exemplo.

```
class Teste {
     public void fazAlgo() {
           System.out.println("Este é o método da super classe");
     }
```

```
}

class NovoTeste : public Teste {
     public void fazAlgo() {
          System.out.println("Este é o método sobrescrito");
     }
}
```

Superposição de métodos

Consideremos uma hierarquia composta de uma classe base, chamada classe *Mamifero*, e uma classe derivada, chamada classe *Cachorro*. Um objeto da classe *Cachorro* tem acesso às funções membro da classe *Mamifero*. Além disso, a classe Cachorro pode acrescentar suas próprias funções membros, como por exemplo, *abanarCauda()*.
A classe cachorro pode ainda superpôr (*override*) uma função da classe base. Superpôr uma função significa mudar a implementação de uma função da classe base na classe derivada. Quando criamos um objeto da classe derivada, a versão correta da função é chamada.
Observe que, para que haja superposição, a nova função deve retornar o mesmo tipo e ter a mesma assinatura da função da classe base. Assinatura, refere-se ao protótipo da função, menos o tipo retornado: ou seja, o nome, a lista de parâmetros e a palavra-chave const, se for usada.

```
//--------------------------------------------------------
// Overrd.cpp
// Apresenta o uso da superposição de métodos (overriding)
#include <iostream.h>
class ClasseBase
{
        protected:
              int m_propr_base1;
              int m_propr_base2;
        public:
// Construtores.
              ClasseBase() : m_propr_base1(10),
              m_propr_base2(20) {}
// Destrutor.
              ~ClasseBase() {}
// Métodos de acesso.
              int acessaPropr1() const {return m_propr_base1;}
              void definePropr1(int valor){ m_propr_base1 = valor;}
              int acessaPropr2() const {return m_propr_base2;}
              void definePropr2(int valor){m_propr_base2 = valor;}
// Outros métodos.
              void met_base1() const
              {
                    cout << "\nEstamos em met_base1...\n";
              } // Fim de met_base1()
              void met_base2() const
              {
                    cout << "\nEstamos em met_base2...\n";
              } // Fim de met_base2()
}; // Fim de class ClasseBase
class ClasseDeriv : public ClasseBase
{
        private:
              int m_propr_deriv;
        public:
// Construtor.
              ClasseDeriv() : m_propr_deriv(1000){}
// Destrutor.
```

```
                ~ClasseDeriv() {};
// Métodos de acesso.
                int acessaPropr_deriv() const
                {
                      return m_propr_deriv;
                } // Fim de acessaPropr_deriv()
                void definePropr_deriv(int valor)
                {
                      m_propr_deriv = valor;
                } // Fim de definePropr_deriv()
// Outros métodos.
                void metodoDeriv1()
                {
                      cout << "Estamos em metodoDeriv1()...\n";
                } // Fim de metodoDeriv1()
                void metodoDeriv2()
                {
                      cout << "Estamos em metodoDeriv2()...\n";
                } // Fim de metodoDeriv2()
// Superpôe (overrides) métodos da classe base.
                void met_base1() /*const*/;
                void met_base2() /*const*/;
}; // Fim de class ClasseDeriv.
// Implementações.
void ClasseDeriv::met_base1() /*const*/
{
        cout << "\nmet_base1() definido na classe derivada...\n";
} // Fim de ClasseDeriv::met_base1()
void ClasseDeriv::met_base2() /*const*/
{
        cout << "\nmet_base2() definido na classe derivada...\n";
} // Fim de ClasseDeriv::met_base2()
int main()
{
// Cria um objeto de ClasseDeriv.
        ClasseDeriv objDeriv;
// Chama métodos superpostos.
        objDeriv.met_base1();
        objDeriv.met_base2();
// Chama métodos da classe derivada.
        objDeriv.metodoDeriv1();
        cout << "Valor de m_propr_deriv = "<< objDeriv.acessaPropr_deriv()
<< "\n";
        return 0;
} // Fim de main()
//------------------------------------------------------
```

Ocultando métodos da classe base

Quando superpomos um método na classe derivada, o método de mesmo nome da classe base fica inacessível. Dizemos que o método da classe base fica oculto. Acontece que muitas vezes, a classe base base tem várias versões sobrecarregadas de um método, com um único nome. Se fizermos a superposição de apenas um desses métodos na classe derivada, todas as outras versões da classe base ficarão inacessíveis. O exemplo abaixo ilustra esse fato.

```
//---------------------------------------------------
// OculMet.cpp
// Ilustra ocultação de métodos da classe base.
#include <iostream.h>
class ClasseBase
```

```
{
        protected:
              int m_propr_base1;
              int m_propr_base2;
        public:
              void met_base() const
              {
                    cout << "\nClasseBase::met_base()...\n";
              } // Fim de met_base1()
              void met_base(int vlr) const
              {
                    cout << "\nClasseBase::met_base(int)...\n";
                    cout << "\nValor = "<< vlr<< "\n";
              } // Fim de met_base1(int)
}; // Fim de class ClasseBase
class ClasseDeriv : public ClasseBase
{
        private:
              int m_propr_deriv;
        public:
// Superpôe (overrides)método da classe base.
              void met_base() const;
}; // Fim de class ClasseDeriv.
// Implementações.
void ClasseDeriv::met_base() const
{
        cout << "\nClasseDeriv::met_base1()...\n";
} // Fim de ClasseDeriv::met_base1()
int main()
{
// Cria um objeto de ClasseDeriv.
        ClasseDeriv objDeriv;
// Chama método superposto.
        objDeriv.met_base();
// Tenta chamar método da classe base.
//objDeriv.met_base(10);
        return 0;
} // Fim de main()
//-----------------------------------------------------------
```

Acessando métodos superpostos da classe base

C++ oferece uma sintaxe para acessar métodos da classe base que tenham ficado ocultos pela superposição na classe derivada. Isso é feito com o chamado operador de resolução de âmbito, representado por dois caracteres de dois pontos ::

Assim, digamos que temos uma classe base chamada *ClasseBase*. *ClasseBase* tem um método *met_base()*, que foi superposto na classe derivada *ClasseDeriv*. Assim, *met_base()* da classe base fica inacessível (oculto) na classe derivada *ClasseDeriv*. Para acessá-lo, usamos a notação:

ClasseBase::met_base();

Por exemplo, se tivermos um objeto de *ClasseDeriv* chamado *objDeriv*, podemos usar a seguinte notação para acessar o método de *ClasseBase*:

objDeriv.ClasseBase::met_base(10);

```
//----------------------------------------------------------
// AcsOcul.cpp
// Ilustra acesso a métodos ocultos na classe base.
#include <iostream.h>
class ClasseBase
{
```

```
        protected:
              int m_propr_base1;
              int m_propr_base2;
        public:
              void met_base() const
              {
                    cout << "\nClasseBase::met_base()...\n";
              } // Fim de met_base1()
              void met_base(int vlr) const
              {
                    cout << "\nClasseBase::met_base(int)...\n";
                    cout << "\nValor = "<< vlr<< "\n";
              } // Fim de met_base1(int)
}; // Fim de class ClasseBase
class ClasseDeriv : public ClasseBase
{
        private:
              int m_propr_deriv;
        public:
// Superpôe (overrides) método da classe base.
        void met_base() const;
}; // Fim de class ClasseDeriv.
// Implementações.
void ClasseDeriv::met_base() const
{
        cout << "\nClasseDeriv::met_base1()...\n";
} // Fim de ClasseDeriv::met_base1()
int main()
{
// Cria um objeto de ClasseDeriv.
        ClasseDeriv objDeriv;
// Chama método superposto.
        objDeriv.met_base();
// Tenta chamar método da classe base.
        objDeriv.ClasseBase::met_base(10);
        return 0;
} // Fim de main()
//------------------------------------------------
```

Métodos virtuais

Até agora, temos enfatizado o fato de que uma hierarquia de herança cria um relacionamento do tipo é um. Por exemplo, um objeto da classe *Cachorro* é um *Mamifero*. Isso significa que o objeto da classe *Cachorro* herda os atributos (dados) e as capacidades (métodos) de sua classe base. Porém, em C++, esse tipo de relacionamento vai ainda mais longe.
Através do polimorfismo, C++ permite que ponteiros para a classe base sejam atribuídos a objetos da classe derivada. Portanto, é perfeitamente legal escrever:

```
Mamifero* pMamifero = new Cachorro;
```

Estamos criando um novo objeto da classe *Cachorro* no *free store*, e atribuindo o ponteiro retornado por *new* a um ponteiro para *Mamifero*. Não há nenhum problema aqui: lembre-se, um *Cachorro* é um *Mamifero*. Podemos usar esse ponteiro para invocar métodos da classe *Mamifero*. Mas seria também desejável poder fazer com que os métodos superpostos em *Cachorro* chamassem a versão correta da função. Isso é possível com o uso de funções virtuais. Veja o exemplo.

```
Exemplo
//--------------------------------------------------------
```

```
// Virt.cpp
// Ilustra o uso de métodos virtuais.
#include <iostream.h>
class Mamifero
{
        protected:
              int m_idade;
      public:
// Construtor.
              Mamifero(): m_idade(1)
              {
                    cout << "Construtor Mamifero()...\n";
              } // Fim de Mamifero()
              ~Mamifero()
              {
                    cout << "Destrutor ~Mamifero()...\n";
              } // Fim de ~Mamifero()
              void andar() const
              {
                    cout << "Mamifero anda 1 passo.\n";
              } // Fim de andar()
// Um método virtual.
              virtual void emiteSom() const
              {
                    cout << "Som de mamifero.\n";
              } // Fim de emiteSom()
}; // Fim de class Mamifero.
class Cachorro : public Mamifero
{
        public:
// Construtor.
              Cachorro() {cout << "Construtor Cachorro()...\n";}
// Destrutor.
              ~Cachorro() {cout << "Destrutor ~Cachorro()...\n";}
              void abanaCauda() { cout << "Abanando cauda...\n";}
// Implementa o método virtual.
              void emiteSom() const {cout << "Au! Au! Au!\n";}
// Implementa outro método.
              void andar() const {cout << "Cachorro anda 5 passos.\n";}
}; // Fim de class Cachorro.
int main()
{
// Um ponteiro para Mamifero
// aponta para um objeto Cachorro.
        Mamifero* pMam = new Cachorro;
// Chama um método superposto.
        pMam->andar();
// Chama o método virtual superposto.
        pMam->emiteSom();
        return 0;
} // Fim de main()
//--------------------------------------------------
```

Chamando múltiplas funções virtuais

Como funcionam as funções virtuais? Quando um objeto derivado, como o objeto *Cachorro*, é criado, primeiro é chamado o construtor da classe base *Mamifero*; depois é chamado o construtor da própria classe derivada *Cachorro*.

Assim, o objeto *Cachorro* contém em si um objeto da classe base *Mamifero*. As duas partes do objeto *Cachorro* ficam armazenadas em porções contíguas da memória. Quando uma função virtual

é criada em um objeto, o objeto deve manter controle sob essa nova função. Muitos compiladores utilizam uma tabela de funções virtuais, chamada *v-table*. Uma *v-table* é mantida para cada tipo, e cada objeto desse tipo mantém um ponteiro para a *v-table*. Esse ponteiro é chamado *vptr*, ou *vpointer*).
Assim, o *vptr* de cada objeto aponta para a *v-table* que, por sua vez, tem um ponteiro para cada uma das funções virtuais. Quando a parte *Mamifero* de um objeto *Cachorro* é criada, o *vptr* é inicializado para apontar para a parte certa da *v-table*. Quando o construtor de *Cachorro* é chamado e a parte *Cachorro* do objeto é acrescentada, o *vptr* é ajustado para apontar para as funções virtuais superpostas, se houver, no objeto *Cachorro*.

Exemplo

```
//---------------------------------------------------------
// MulVirt.cpp
// Ilustra chamada a múltiplas versões de um método virtual.
#include <iostream.h>
class Mamifero
{
        protected:
              int idade;
        public:
// Construtor.
              Mamifero() : idade(1) { }
// Destrutor.
              ~Mamifero() {}
// Método virtual.
              virtual void emiteSom() const
              {
                    cout << "Som de mamifero.\n";
            } // Fim de emiteSom()
}; // Fim de class Mamifero.
class Cachorro : public Mamifero
{
        public:
// Implementa método virtual.
              void emiteSom() const {cout << "Au! Au!\n";}
}; // Fim de class Cachorro.
class Gato : public Mamifero
{
        public:
// Implementa método virtual.
              void emiteSom() const {cout << "Miau!\n";}
}; // Fim de class Gato
class Cavalo : public Mamifero
{
        public:
// Implementa método virtual.
              void emiteSom() const {cout << "Relincho!\n";}
}; // Fim de class Cavalo.
class Porco : public Mamifero
{
        public:
// Implementa método virtual.
              void emiteSom() const {cout << "Oinc!\n";}
}; // Fim de class Porco.
int main()
{
// Um ponteiro para Mamifero.
        Mamifero* mamPtr;
        int opcao;
```

```
        bool flag = true;
        while(flag)
        {
              cout << "\n(1)Cachorro"<< "\n(2)Gato"<< "\n(3)Cavalo"
<< "\n(4)Porco"<< "\n(5)Mamifero";
              cout << "\nDigite um num. ou "<< "zero para sair: ";
              cin >> opcao;
              switch(opcao)
              {
                    case 0:
                          flag = false;
                          break;
                    case 1:
                          mamPtr = new Cachorro;
                          mamPtr->emiteSom();
                          break;
                    case 2:
                          mamPtr = new Gato;
                          mamPtr->emiteSom();
                          break;
                    case 3:
                          mamPtr = new Cavalo;
                          mamPtr->emiteSom();
                          break;
                    case 4:
                          mamPtr = new Porco;
                          mamPtr->emiteSom();
                          break;
                    case 5:
                          mamPtr = new Mamifero;
                          mamPtr->emiteSom();
                          break;
                    default:
                          cout << "\nOpcao invalida.";
                          break;
              } // Fim de switch
        } // Fim de while.
        return 0;
} // Fim de main()
//-----------------------------------------------------
```

Métodos virtuais e passagem por valor

Observe que a mágica da função virtual somente opera com ponteiros ou referências. A passagem de um objeto por valor não permite que funções virtuais sejam invocadas. Veja o exemplo abaixo.

```
//--------------------------------------------------------------------
// VirtVal.cpp
// Ilustra tentativa de usar métodos virtuais com argumento passado
// por valor.
#include <iostream.h>
class Mamifero
{
        protected:
              int idade;
        public:
// Construtor.
              Mamifero() : idade(1) { }
// Destrutor.
              ~Mamifero() {}
```

```
// Método virtual.
              virtual void emiteSom() const
              {
                    cout << "Som de mamifero.\n";
              } // Fim de emiteSom()
}; // Fim de class Mamifero.
class Cachorro : public Mamifero
{
        public:
// Implementa método virtual.
              void emiteSom() const
              {
                    cout << "Au! Au!\n";
              } // Fim de emiteSom()
}; // Fim de class Cachorro.
class Gato : public Mamifero
{
        public:
// Implementa método virtual.
              void emiteSom() const
              {
                    cout << "Miau!\n";
              } // Fim de emiteSom()
}; // Fim de class Gato.
// Protótipos.
void funcaoPorValor(Mamifero);
void funcaoPorPonteiro(Mamifero*);
void funcaoPorRef(Mamifero&);
int main()
{
        Mamifero* mamPtr;
        int opcao;
        cout << "\n(1)Cachorro"<< "\n(2)Gato"<< "\n(3)Mamifero"<<
"\n(0)Sair";
        cout << "\n\nEscolha uma opcao: ";
        cin >> opcao;
        cout << "\n";
        switch(opcao)
        {
              case 0:
                    mamPtr = 0;
                    break;
              case 1:
                    mamPtr = new Cachorro;
                    break;
              case 2:
                    mamPtr = new Gato;
                    break;
              case 3:
                    mamPtr = new Mamifero;
                    break;
              default:
                    cout << "\nOpcao invalida.\n";
                    mamPtr = 0;
                    break;
        } // Fim de switch.
// Chama funções.
        if(mamPtr)
        {
              funcaoPorPonteiro(mamPtr);
              funcaoPorRef(*mamPtr);
```

```
                funcaoPorValor(*mamPtr);
        } // Fim de if(mamPtr)
        return 0;
} // Fim de main()
void funcaoPorValor(Mamifero mamValor)
{
        mamValor.emiteSom();
} // Fim de funcaoPorValor()
void funcaoPorPonteiro(Mamifero* pMam)
{
        pMam->emiteSom();
} // Fim de funcaoPorPonteiro()
void funcaoPorRef(Mamifero& refMam)
{
        refMam.emiteSom();
} // Fim de funcaoPorRef()
//------------------------------------------------------
```

Construtor de cópia virtual

Métodos construtores não podem ser virtuais. Contudo, há ocasiões em que surge uma necessidade de poder passar um ponteiro para um objeto base e fazer com que uma cópia do objeto derivado correto seja criado. Uma solução comum para esse problema é criar um método chamado clone() na classe base e torná-lo virtual. O método clone() cria uma cópia do novo objeto da classe atual, e retorna esse objeto. Como cada classe derivada superpõe o método clone(), uma cópia do objeto correto é criada.

```
Exemplo
//--------------------------------------------------------
// VirtCop.cpp
// Ilustra uso do método clone() como substituto para
// um construtor de cópia virtual.
#include <iostream.h>
class Mamifero
{
        public:
              Mamifero() : idade(1)
              {
                    cout << "Construtor de Mamifero...\n";
              } // Fim de Mamifero()
              ~Mamifero()
              {
                    cout << "Destrutor de Mamifero...\n";
              } // Fim de ~Mamifero()
// Construtor de cópia.
              Mamifero(const Mamifero& refMam);
// Métodos virtuais.
              virtual void emiteSom() const
              {
                    cout << "Som de mamifero.\n";
              } // Fim de emiteSom()
              virtual Mamifero* clone()
              {
                    return new Mamifero(*this);
              } // Fim de clone()
              int acessaIdade() const
              {
                    return idade;
              } // Fim de acessaIdade()
        protected:
```

```
        int idade;
}; // Fim de class Mamifero.
// Construtor de cópia.
Mamifero::Mamifero(const Mamifero& refMam) :idade(refMam.acessaIdade())
{
        cout << "Construtor Mamifero(Mamifero&)...\n";
} // Fim de Mamifero::Mamifero(const Mamifero&)
class Cachorro : public Mamifero
{
        public:
              Cachorro()
              {
                    cout << "Construtor Cachorro()...\n";
              } // Fim de Cachorro()
              ~Cachorro()
              {
                    cout << "Destrutor ~Cachorro()...\n";
              } // Fim de ~Cachorro()
// Construtor de cópia.
              Cachorro(const Cachorro& refCach);
// Implementa métodos virtuais.
              void emiteSom() const
              {
                    cout << "Au!Au!\n";
              } // Fim de emiteSom()
              virtual Mamifero* clone()
              {
                    return new Cachorro(*this);
              } // Fim de clone()
}; // Fim de class Cachorro.
// Construtor de cópia.
Cachorro::Cachorro(const Cachorro& refCach) :Mamifero(refCach)
{
        cout << "Construtor Cachorro(Cachorro&)...\n";
} // Fim de Cachorro::Cachorro(Cachorro&)
class Gato : public Mamifero
{
        public:
              Gato()
              {
                    cout << "Construtor Gato()...\n";
              } // Fim de Gato()
              ~Gato()
              {
                    cout << "Destrutor ~Gato()...\n";
              } // Fim de ~Gato()
// Construtor de cópia.
              Gato(const Gato& refGato);
// Implementa métodos virtuais.
              void emiteSom() const
              {
                    cout << "Miau!\n";
              } // Fim de emiteSom()
              virtual Mamifero* clone()
              {
                    return new Gato(*this);
              } // Fim de clone()
}; // Fim de class Gato.
// Construtor de cópia.
Gato::Gato(const Gato& refGato) :Mamifero(refGato)
{
```

```
        cout << "Construtor Gato(const Gato&)...\n";
} // Fim de Gato::Gato(const Gato&)
enum ANIMAIS {MAMIFERO, CACHORRO, GATO};
int main()
{
// Um ponteiro para Mamifero.
        Mamifero* mamPtr;
        int opcao;
// Exibe menu.
        cout <<"\n(1)Cachorro"<< "\n(2)Gato"<< "\n(3)Mamifero\n";
        cout << "\nDigite a opcao: ";
        cin >> opcao;
        switch(opcao)
        {
              case CACHORRO:
                    mamPtr = new Cachorro;
                    break;
              case GATO:
                    mamPtr = new Gato;
                    break;
              default:
                    mamPtr = new Mamifero;
                    break;
        } // Fim de switch.
// Um outro ponteiro para Mamifero.
        Mamifero* mamPtr2;
        cout << "\n*** Som do original ***\n";
// Emite som.
        mamPtr->emiteSom();
// Cria clone.
        mamPtr2 = mamPtr->clone();
        cout << "\n*** Som do clone ***\n";
        mamPtr2->emiteSom();
        return 0;
} // Fim de main()
//----------------------------------------------------
```

Herança múltipla

Em C++, a herança não se limita a uma única classe base. Uma classe pode ter vários pais, herdando características de todos eles. Este tipo de herança introduz grande dose de complexidade na linguagem e no compilador, mas os benefícios são substanciais. Considere a criação de uma classe *MesaRedonda*, tendo não só propriedades de mesas, mas também as características geométricas de ser redonda. O código abaixo é uma possível implementação:

```
class Circulo {
        float raio;
        public:
              Circulo(float r) { raio = r; }
              float area() { return raio*raio*3.14159; }
};
class Mesa {
        float ipeso;
        float ialtura;
        public:
              Mesa(float p, float a) { ipeso = p; ialtura=a; }
              float peso() { return ipeso; }
              float altura() { return ialtura; }
};
class MesaRedonda: public Circulo, public Mesa {
```

```
        int icor;
        public:
              MesaRedonda(int c, float a, float p, float r);
              int cor() { return icor; }
};
MesaRedonda::MesaRedonda(int c, float a, float p, float r): Mesa(p, a),
Circulo(r)
{
        icor = c;
}
void main()
{
        MesaRedonda mesa(5, 1, 20, 3.5 );
        printf("Peso: %f\n", mesa.peso());
        printf("Altura: %f\n", mesa.altura());
        printf("Area: %f\n", mesa.area());
        printf("Cor: %d\n", mesa.cor());
};
```

Um exemplo natural poderia sair da seção que discute herança de tipo ou código. Para implementar uma pilha com listas encadeadas que possa ser usada como Stack e aproveitando uma classe já implementada de listas encadeadas, a declaração seria assim:

```
class StackList : public Stack, private LinkedList {
// ...
};
```

Ordem de chamada dos construtores e destrutores

Assim como em herança simples, os construtores das classes base são chamados antes do construtor da classe derivada. A ordem de declaração na classe define a ordem de chamada dos construtores. No exemplo acima, a classe foi declarada com uma ordem

class MesaRedonda: public Circulo, public Mesa {

e o construtor com outra:

MesaRedonda::MesaRedonda(int c, float a, float p, float r) : Mesa(p, a), Circulo(r)

Como a ordem de declaração na classe é a que define, a ordem dos construtores será:

- Circulo::Circulo
- Mesa::Mesa
- MesaRedonda::MesaRedonda

Classes básicas virtuais

Classes base virtuais <u>só são usadas com herança múltipla</u>. É uma maneira de o programador controlar como as classes devem ser herdadas. Por exemplo:

```
class A {
        public:
              int a;
};
class B: public A {};
class C: public A{};
class D: public B, public C {
```

```
        public:
              int valor() { return a; }
};
```
Este código gera uma hierarquia onde a classe D tem duas cópias da parte A, uma associada a B e outra a C. Portanto, o código acima gera um erro de compilação:
Member is ambiguous: ‘A::a’ and ‘A::a’

O problema é que a declaração de B herdando de A faz com que a classe B já tenha uma “parte A” incorporada. Nesse caso, B já tem a sua variável a. O mesmo acontece com C. O resultado são duas cópias de A na hierarquia. Uma é a “parte A”de B e outra é a “parte A”de C. O compilador não sabe que cópia de a esta sendo referenciada. O operador de escopo poderia ser utilizado para retirar o erro:
int valor() { return C::a; }

Às vezes o programador quer montar uma árvore onde só exista uma cópia de A. É o caso de usar uma classe base virtual. Declarando uma classe base como virtual faz com que a classe derivada não inclua a classe base. Seria o caso de declarar as heranças de B e C como virtuais, assim nenhuma das duas teria uma “parte A”:

```
class B: public virtual A {};
class C: public virtual A{};
class D: public B, public C {
        public:
              int valor() { return a; }
};
```

Agora a função valor não precisa mais do operador de escopo, e a árvore gerada terá apenas uma cópia de A.

Chamada de construtores de classes básicas virtuais

Como na herança virtual as classes derivadas não contém a parte da sua classe base, é preciso tomar alguns cuidados na hora de inicializar estas classes básicas. Supondo que a classe A tenha um construtor que receba um inteiro:

```
class A {
        public:
              int a;
              A(int) {}
};
```

A chamada a este construtor tem que estar explícita no código de B e C:

```
class B: public virtual A { public: B() : A(1) {} };
class C: public virtual A { public: C() : A(2) {} };
class D: public B, public C {
        public:
              int valor() { return a; }
// erro! compilador não gera construtor vazio
};
```

Se um objeto da classe B for criado, sua parte A será inicializada com 1. Se for criado um do tipo C, a inicialização será com 2. E se o objeto for do tipo D? Como a parte A é criada diretamente por D (herança virtual), o próprio construtor de D deve chamar o de A diretamente:

```
class D: public B, public C {
```

```
        public:
              int valor() { return a; }
              D(): A(3) {}
};
```

Programação Orientada por Objetos – Tópico 7

***Temática**: Biblioteca streams, templates e tratamento de exceções*

Actividade 7: Familiarização com o templates e tratamento de exceções

Competências a desenvolver:

- Compreender o conceito de templates e tratamento de exceções
- Aprender a criar templates e tratar exceções
- Aprender a utilizar a biblioteca de streams

Biblioteca de Streams

A linguagem oferece, como biblioteca padrão, um conjunto de classes para tratamento de entrada e saída. Entre as vantagens está a possibilidade de extensão para tratamento de tipos definidos pelo utilizador e notação uniforme para todos os dispositivos.

Stream I/O

Streams trazem a elegância da sobrecarga de operadores para a parte de I/O. O objetivo das *streams* é uniformizar a notação dentre os diferentes tipos de I/O, deixando os detalhes para serem resolvidos pelo compilador. O uso de *streams* tem a seguinte notação:

```
stream_de_entrada >> variável;
stream_de_saída << variável;
```

O objeto *stream* é sempre colocado à esquerda na expressão. Os operadores << e >> são utilizados para indicar o fluxo de dados de um objeto para outro. Todos os tipos pré-definidos podem ser utilizados com *streams* de I/O. Classes definidas pelo usuário também podem usar *streams* se as classes suportarem estas operações de I/O. O exemplo abaixo ilustra o uso de *streams*:

```
#include <iostream.h>
void main()
{
        int a;
        char c;
        float f;
        double d;
        cin >> a;
        cin >> c;
        cin >> f >> d;
        cout << a;
        cout << c << f << d;
        cout << "string";
}
```

C++ mantém a definição de entrada e saída de dados padrão, mas usa *streams* no lugar de arquivos. A *stream cin* é a entrada padrão, enquanto que *cout* é a saída padrão. Estas *streams* substituem *stdin* e *stdout* repectivamente, e são usadas com o mesmo propósito. Existem ainda duas streams *cerr* (semelhante a *stderr*) e *clog*, que não tem semelhante em ANSI C.
Assim como *stdin*, *stdout* e *stderr*, estas *streams* não precisam ser declaradas, inicializadas ou destruídas. Para usá-las, basta incluir o arquivo *iostream.h*. *Streams* não são mais do que uma biblioteca; ou seja, é um código normal como outro qualquer, que não precisa de nenhum tratamento especial por parte do compilador.
O arquivo *iostream.h* define algumas classes, entre elas *istream*, *ostream* e *iostream*. *istream* é uma

stream só de entrada, *ostream* só de saída e *iostream* é uma classe que herda de *istream* e *ostream*, servindo tanto para entrada como para saída. *cin* é um objeto de um tipo derivado de *istream*, que é declarado *extern* no arquivo *iostream.h*:

extern istream_withassign cin;

cin então é um objeto global, existindo durante todo o programa. Sendo global, o seu construtor é executado antes de *main* e o destrutor, depois de *main*. O construtor associa *cin* com a entrada padrão, e o destrutor desfaz esta associação. Daí não ser necessário se preocupar com a declaração, inicialização ou destruição.
cin funciona exclusivamente através de operadores. A classe *istream* define vários operadores >>, cada um recebendo um tipo diferente de parâmetro e retornando o próprio objeto. O que acontece no comando:

cin >> a;

é simplesmente a aplicação do método *operator>>* sobre o objeto *cin* passando como parâmetro a variável *a*. Já no seguinte caso:

cin >> a >> b;

o que acontece? É o mesmo caso de uma expressão a + b + c; Primeiro, a expressão a + b é avaliada; depois, o resultado é somado a c, resultando em (a + b) + c. Como o método operator>> retorna o próprio objeto (*cin*), a linha acima é o mesmo que *(cin >> a) >> b*; O mesmo acontece com *cout*.

I/O com classes definidas pelo utilizador

As classes de streams permitem que classes definidas pelo usuário usem a mesma notação para qualquer tipo pré-definido. Basta sobrecarregar os operadores << e >>.
Vamos ver como estes operadores devem ser sobrecarregados. Supondo que a classe Ponto tenha os operadores, um uso típico seria:

int a, b;
Ponto p;
cin >> a >> p >> b;

Primeiro, será executado *cin* >> a. Ao resultado desta expressão (*cin*) será aplicado o *operador* >>, resultando em *cin* >> p. Fazendo esta análise, notamos que o operador não pode ser um membro na classe *Ponto*, já que o operador é aplicado a *cin*, e não a *p*. O operador deve ser definido assim:

```
istream& operator>> (istream& is, Ponto& p);
```

Eis a declaração completa de *Ponto*:

```
class Ponto {
        float x, y, z;
        public:
              Ponto(float a, float b, float c){ x=a; y=b; z=c; }
              friend ostream& operator<< (ostream& os, Ponto& p);
              friend istream& operator>> (istream& is, Ponto& p);
};
ostream& operator<< (ostream& os, Ponto& p)
{
        return os << '(' << p.x << ','<< p.y << ','<< p.z << ')';
}
istream& operator>> (istream& is, Ponto& p)
{
        return is >> p.x >> p.y >> p.z;
```

```
}
```

Além dos operadores << e >>, existem algumas funções. Por exemplo, a classe istream tem o métodos
istream& istream::putback(char);
istream& istream::getline(char*, int, char = '\n');

Manipuladores

A biblioteca ANSI C permite uma certa flexibilidade no modo como os dados podem ser formatados. Com *streams* isto é feito através de manipuladores. Manipuladores são funções especialmente designadas para modificar o modo como uma *stream* trabalha. O arquivo *iostream.h* vem com uma série de manipuladores, e o arquivo *iomanip.h* define mais alguns.
Aqui está a lista dos manipuladores pré-definidos:

- dec: mostra os números na base decimal. Afeta int e long.
- hex: base hexadecimal
- oct: base octal
- ws: extrai brancos de uma istream
- endl: insere um caractere de fim de linha
- ends: insere um caractere de fim de string
- flush: decarrega os buffers de saída
- setbase(int): modifica a base. Aceita os valores 0 (default, =10), 8, 10 e 16.
- resetiosflags(long): limpa um ou mais flags de ios::x_flags
- setioflags(long): seta um ou mais flags de ios::x_flags
- setfill(int): define o caractere usado para preencher caracteres não usados quando a largura mínima é maior do que a utilizada; ver setw
- setprecision(int): define o número de dígitos decimais para float e double
- setw(int): define a largura da próxima variável a ser colocada em uma stream de saída.

Estes modificadores são usados assim:

```
cout << hex << 10 << setfill('.') << setwidth(10) << dec << 23;
```

A linha acima gera a seguinte saída:

```
a........23
```

Arquivos de entrada como streams

A elegância e a simplicidade das *streams* também pode ser utilizada para arquivos, tanto no modo texto como no binário. As classes utilizadas são *ifstream* e *ofstream*. Arquivos são utilizados da mesma maneira que a apresentada nas seções anteriores, a única diferença é na hora da criação. Os construtores recebem o nome do arquivo a ser aberto:

```
#include <iostream.h>
#include <fstream.h>
void main()
{
        ifstream file("teste.c");
        if (!file)
        {
              return;
        }
        while (file)
        {
              char buffer[100];
              file.getline(buffer, 100);
```

```
            cout << endl << buffer;
      }
}
```

Existe um parâmetro default no construtor que não foi utilizado no exemplo acima, que serve para indicar que o arquivo é binário:

ifstream file(“teste.c”, ios::binary);

Testando erros em uma stream

Durante as operações, pode ocorrer uma situação de erro em uma stream. Pode-se tentar abrir um arquivo inexistente, ler depois do fim do arquivo etc. Todas estas condições causam erros, e devem ser detectadas e tratadas. A maneira mais simples de testar se ocorreu um erro é usando expressões da forma:

```
if (!file) // ocorreu um erro
```

ou

```
if (file) // nenhum erro
```

Expressões deste tipo são possíveis porque os operadores ! e *void** são sobrecarregados em *streams*. Existem maneiras de investigar a causa do erro, como a função *rdstate()*, que retorna o tipo do erro. Outras funções estão disponíveis, como

```
if (file.bad()) // erro
if (file.eof()) // fim de arquivo
if (file.good()) // nenhum erro
```

Arquivos de saída como streams

O uso de arquivos de saída é semelhante aos de entrada. A diferença é que o parâmetro opcional do construtor pode receber outros valores além de ios::binary, e indicam o modo de abrir o arquivo:

```
// abre um arquivo para escrita, apaga se já existe
ofstream file1("teste.out");
// abre um arquivo para escrita no fim
ofstream file2("teste.out",ios::app);
// abre um arquivo vazio para escrita, se já existe gera erro
ofstream file3("teste.out",ios::noreplace);
// abre um arquivo vazio para escrita, erro se não existe
ofstream file4("teste.out",ios::nocreate);
```

Formatação na memória

Muitas vezes é preciso formatar dados sem escrever em um arquivo ou na saída padrão. Em ANSI C isto é feito com a função *sprintf*. A classe *strstream* pode ser usada para este fim. Ela funciona como qualquer stream com manipuladores etc. A classe *istrstream* tem a finalidade de ler ler uma string e extrair dados dela.
Exemplos de *strstream*:

```
#include <iostream.h>
#include <strstream.h>
void main()
{
        strstream buffer;
        int a = 20;
        float pi = 3.14159;
        buffer << “O número PI é “ << pi << ‘.’ << endl;
```

```
        buffer << "vinte (" << a << ")" << endl;
        cout << buffer.rdbuf();
}
```

Exemplos de *istrstream*:
```
#include <iostream.h>
#include <strstream.h>
void main(int argc, char *argv[])
{
        istrstream arg(argv[0]);
        cout << "Existem " << argc<< "argumentos. O primeiro e "<<
arg.rdbuf();
}
```

Templates

Uma classe C++ normalmente é projetada para armazenar algum tipo de dado. Muitas vezes a funcionalidade de uma classe também faz sentido com outros tipos de dados. Este é o caso de muitos exemplos apresentados (por exemplo, Pilha).
Se uma classe é vista simplesmente como um manipulador de dados, pode fazer sentido separar a definição desta classe da definição dos tipos manipulados; isto é, pode-se fazer uma descrição de uma classe sem definir o tipo dos dados que ela manipula. Nesse caso, definição da classe é parametrizada por um tipo genérico T, sendo chamada C<T>. Esta construção não é uma classe realmente, mas uma descrição do conjunto de classes com o mesmo comportamento e que operam sobre um tipo T qualquer. Esta construção é denominada *class template*.
Com *class templates*, é permitido ao programador criar um série de classes distintas com a mesma descrição, mas sobre tipos distintos. Por exemplo, pode-se construir uma pilha de inteiros (Pilha<int>) ou pilha de strings (Pilha<char*>) a partir da mesma descrição. Esta descrição abstrata da *template* é utilizada pelo compilador para criar uma classe real em tempo de compilação, usando o tipo dos dados especificados quando de seu uso.
Em alguns textos acadêmicos, esta funcionalidade é chamada de polimorfismo paramétrico.

Declaração de templates

Consideremos uma classe Pilha. Ela serve para gerir dados colocados numa pilha (é uma estrutura de dados especial: http://pt.wikipedia.org/wiki/Pilha_(inform%C3%A1tica)). Ao invés de escrevermos uma nova classe pilha para cada novo tipo demandado, pode-se definir uma *template* para a classe Pilha, onde o tipo armazenado é um T qualquer. A forma desta declaração é mostrada abaixo:

```
template<class T> class Pilha {
        struct elemPilha {
              elemPilha* prox;
              T val;
              elemPilha(elemPilha*p, T v) { prox=p; val=v; }
        };
        elemPilha* topo;
        public:
              int vazia() { return topo == NULL }
              void push(T v) { topo = new elemPilha(topo, v); }
              T pop(){
                    if (topo)
                    {
                          elemPilha *ep = topo;
                          T v = ep->val;
                          topo = ep->prox;
                          delete ep;
```

```
                    return v;
                }
                return -1;
            }
};
```

Tendo em vista que uma *template* é simplesmente uma descrição de uma classe, é necessário que toda esta descrição tenha sido lida antes de alguma declaração que envolva esta *template*. Isto significa, em se tratando de *templates*, que é necessário colocar no arquivo .h não apenas a declaração de classe, mas também a implementação de seus métodos.
Outra consequência de *templates* serem apenas descrições é que erros semânticos só aparecem na hora de usar a *template*. Durante a declaração da *template* apenas erros de sintaxe são verificados. Mesmo que a *template* em si tenha sido compilada sem erros, podem aparecer erros quando de sua utilização. Esta verificação semântica é realizada todas as vezes que a *template* é instanciada para algum tipo novo. Isto porque na definição do código da *template* não há nenhuma restrição quanto às operações que podem ser aplicadas ao tipo T. A verificação então tem que ser feita para cada tipo.

Usando templates

Definida a implementação de nossa *template* de pilhas, pode-se utilizá-la para qualquer tipo T criando pilhas de inteiros, strings,etc. Na criação de objetos destas classes pilhas, é necessário especificar o tipo de pilha na declaração do objeto:

```
void main()
{
        Pilha<int> intPilha; // pilha de inteiros
        Pilha<char*> stringPilha; // pilha de strings
        Pilha<Pilha<int>> intPilhaPilha; // pilha de pilha de inteiros
        intPilha.push(10);
        stringPilha.push( "teste" );
        intPilhaPilha.push( intPilha );
}
```

Declaração de métodos não inline

Quando da declaração de uma *template*, não é obrigatoria a implementação de seus métodos na forma *inline*; isto é, sua implementação pode não estar no corpo da declaração da classe. Para tal, basta especificar, de maneira análoga a descrição de classes, a *template* a que o método pertence. Segue abaixo a implementação do método *pop* fora do corpo da *template*:

```
template<class T> class Pilha {
// estrutura interna da template...
        public:
              int vazia() { return topo == NULL }
              void push(T v) { topo = new elemPilha(topo, v); }
              T pop(); // protótipo do método pop
// implementado abaixo (fora da classe)
};// Fim da declaração da classe
template<class T> T Pilha<T>::pop(){
        if (topo){
              elemPilha *ep = topo;
              T v = ep->val;
              topo = ep->prox;
              delete ep;
              return v;
        }
        return -1;
}
```

Templates com vários argumentos genéricos

Templates não estão limitadas a terem apenas um único argumento. Pode-se utilizar diversos argumentos para parametrizar a descrição da classe. No exemplo abaixo, a *Pilha* armazena dois tipos de dados e ambos são parâmetros da *template*:

```
template<class T, class R> class Pilha {
        struct elemPilha {
              elemPilha* prox;
              T t_val;
              R r_val;
              elemPilha(elemPilha*p, T t, R r)
              { prox=p; t_val=t; r_val=r; }
        };
        elemPilha* topo;
        public:
              int vazia() { return topo == NULL }
              void push(T t, R r) { topo = new elemPilha(topo, t, r); }
              void pop(T& t, R& r);
};
```

Templates com argumentos não genéricos

Templates não estão limitadas a argumentos genéricos; ou seja, tipos definidos pelo programador. Pode-se utilizar como argumentos da *template* tipos primitivos da linguagem como inteiros ou caracteres. No exemplo abaixo, tem-se uma pilha que armazena um número fixo de elementos de um mesmo tipo:

```
template<class T, int S> class Pilha {
        struct elemPilha {
              elemPilha* prox;
              T t_val[S];
              elemPilha(elemPilha*p, T* t);
        };
        elemPilha* topo;
        public:
              int vazia();
              void push(T* t);
              void pop(T* t);
};
```

Templates de funções

Templates também podem ser usadas para definir funções. A mesma motivação para classes vale neste caso. Algumas vezes funções realizam operações sobre dados sem utilizar o conteúdo deles, ou seja, indepentemente de que tipo de dado seja.
Suponha que precisamos testar a magnitude de dois elementos quaisquer. A seguinte macro resolve este problema:

```
#define max(a,b) ((x>y) ? x : y)
```

Por muito tempo macros como esta foram usadas em programas C, mas isto tem os seus problemas. A macro funciona, mas impede que o compilador teste os tipos dos elementos envolvidos. A macro poderia ser utilizada para comparar um inteiro com um ponteiro, sem que sejam gerados erros. Poderíamos usar uma função como esta:

```
int max(int a, int b)
{
        return a > b ? a : b;
}
```

Mas esta função só funciona para inteiros. Se o nosso programa só trabalha com escalares, poderíamos escrever uma função que trabalhe com *double*:

```
double max(double a, double b)
{
        return a > b ? a : b;
}
```

Nesse caso, o compilador se encarrega de converter os tipos *char*, *int* etc. para *double*, e a função funcionaria para todos estes casos. Existem pelo menos duas limitações nesta versão. Uma delas diz respeito ao tipo dos parâmetros: apenas tipos que podem ser convertidos para *double* podem usar esta função. Isto significa que objetos e ponteiros não podem ser utilizados. A outra limitação se refere ao tipo de retorno: este é sempre *double*, independente do tipo passado. Suponha que a função *display* seja sobrecarregada para imprimir vários tipos de dados. Agora considere o código:

```
display(max('1', '9'));
```

Apesar de estarmos trabalhando com caracteres, a função display a ser chamada será a versão que trabalha com *double*, e o resultado será 57.00, que é o código ASCII do caractere '9'. A opção de sobrecarregar max para vários tipos também não é boa, pois teríamos que reescrever o código, que seria idêntico, para todos os tipos que quiséssemos usar. Ainda assim o problema não estaria resolvido, pois novos tipos não poderiam ser usados sem que se criasse outra versão sobrecarregada de *max*.
Na verdade, qualquer tipo de dado que possua operações de comparação pode ser usado com *max*, e sempre da mesma maneira. Não é possível escrever uma única função que trate todos os tipos, mas o mecanismo de *templates* possibilita descrever, para o compilador, como estas funções podem ser implementadas:

```
template<class T> T max( T a, T b )
{
        return a > b ? a : b;
}
```
Esta função faz sentido para qualquer tipo T. Na realidade existirá uma implementação para cada tipo que for usado com esta função dentro do programa, mas estas funções serão geradas transparentemente pelo compilador. O uso de *templates* de funções não exige que se especifique explicitamente os tipos genéricos na chamada, como em *templates* de classes. Basta usar como se existisse uma função específica para o tipo envolvido:

```
void main()
{
        printf("%c", max('1', '9'));
}
```

Repare que, apesar da função ser usada para comparar caracteres, em nenhum momento aparece a palavra *char*. O compilador sabe qual max precisa ser chamada pelo tipo dos parâmetros usados. Por causa disto, os tipos genéricos de *templates* de funções devem sempre aparecer nos parâmetros da função.
Caso isto não aconteça, será sinalizado um erro de sintaxe na linha da declaração da *template*. Assim como em *templates* de classes, *templates* de funções podem ter vários parâmetros genéricos.

Tratamento de Exceções

Quando do desenvolvimento de bibliotecas, é possível escrever código capaz de detectar erros de execução mas, em geral, não é possível fazer seu tratamento. Por outro lado, o utilizador de uma biblioteca é capaz de fazer o correto tratamento de uma exceção mas não é capaz de detectá-la.
O conceito de exceção é introduzido para ajudar neste tipo de problema. A idéia fundamental é que uma função que detecte um problema e não seja capaz de resolvê-lo "acuse a exceção" esperando que quem a chamou seja capaz de realizar o tratamento adequado.

Funcionamento básico

O funcionamento dos tratadores de exceção é composto de diversas etapas:

- Definição das exceções que uma classe pode levantar;
- Definição de quando uma classe acusa uma exceção;
- Definição do(s) tratador(res) das exceções.

A definição da execeções que podem ser levantadas é feita na construção da classe. Neste momento define-se as condições de erro e os representa sob a forma de uma classe. Desta forma, cada condição de exceção é descrita por uma classe de exceção. Por exemplo, considere como representar e tratar o erro de indexação fora dos limites de um array dada pela classe *Vector*:

```
class Vector {
        int* p;
        int sz;
        public:
              class Range{ }; // classe de tratamento de exceção que
// representa acesso com índice inválido
              int& operator[]( int i );
};
```

A classe *Range*, a princípio sem dados internos, representa a condição de erro para acesso com índice não válido (menor que zero, maior que o espaço alocado, etc.). A definição dos momentos da exceção são também definidos na construção da classe. A acusação das exceções é feita normalmente nos métodos da classe que encontrem alguma situação de erro. O levantamento de uma exceção é feita pelo comando *throw* que, conforme no nosso exemplo, é acionado quando o índice do array é inválido. Nestas situações, os objetos da classe *Range* são utilizados como exceções e são acusados da seguinte forma:

```
int& Vector::operator[]( int i )
{
        if ( i>=0 && i<sz ) return p[i];
        throw Range(); // Range() cria um objeto da classe Range
// Quando indexamos o vetor com limites inválidos é
// acusada a exceção correspondente
// se a função que chamou operator[] souber tratá-la,
// teremos o tratamento adequado.
}
```

A definição dos tratadores aparecem, normalmente, nas funções que utilizam serviços das classes que levantam exceções. No caso de C++, estas funções podem selecionar quais as exceções que serão tratadas e trechos onde estas podem surgir. No nosso exemplo, a função que precisa detectar a utilização de índices fora do limite indica seu interesse pelo tratamento colocando código correspondente na seguinte forma:

```
void f( Vector& v )
{
//.. código qualquer sem tratamento
        try{
```

```
//.. código qualquer com tratamento
                operação_qualquer( v );
        }
        catch( Vector::Range ) {
// Aqui se encerra o código de tratamento da exceção Range.
// A função operação_qualquer apresentou a exceção que está sendo tratada.
// Esse código somente será executado se e somente se
// a operação_qualquer fizer uso de indexação inválida.
        }
//.. código qualquer sem tratamento
}
```

A construção **catch(/* nome da exceção */) { /* código */ }** é denominada manipulador de exceção (*exception handler*). Esta construção somente pode ser utilizada depois de um bloco prefixado com a palavra reservada *try* ou após outra construção *catch*. Os parênteses encerram a declaração dos tipos de objetos aonde o manipulador pode executar. Se a função operação_qualquer ou qualquer outra função chamada por operação_qualquer causar uma indexação inválida no array, será gerada uma exceção que será pega pelo manipulador e seu código executado.
Se um manipulador pegou uma determinada exceção, esta foi devidamente tratada e qualquer outro manipulador ainda existente se torna irrelevante. Em outras palavras, apenas o manipulador mais recentemente encontrado pelo controle da linguagem será executado. Por exemplo, dado que a função f pega uma exceção *Vector::Range*, uma função que chame *f* jamais pegará a exceção *Vector::Range*.

```
int ff( Vector& v )
{
        try{
                f(v);
        }
        catch( Vector::Range )
        { // este código jamais será executado ...
        }
}
```

Naturalmente, um programa é capaz de tratar diversas exceções. Esses erros são mapeados com nomes distintos. Continuando o exemplo de *Vector*, trataremos mais um caso: criação de *array* com tamanho fora dos limites. Temos:

```
class Vector {
        int* p;
        int sz;
        int max 512; // número máximo de elementos
        public:
                class Range{ }; //classe de tratamento de exceção
                class Size{ }; //classe de tratamento de exceção
                int& operator[]( int i );
                Vector( int sz )
                {
                        if ( sz < 0 || max < sz ) throw Size();
// continuação do construtor...
                }
};
```

O usuário da classe *Vector* pode discriminar a exceção pondo diversos manipuladores dentro do bloco precedido por *try*:

```
void f( Vector& v )
```

```
{
        try {
              qualquer_operação(v);
        }
        catch( Vector::Range ) {
// código de tratamento de indexação inválida
        }
        catch( Vector::Size ) {
// código de tratamento para criação de vetor muito grande
        }
// esse código é executado se não tiver ocorrido nenhuma
// exceção.
}
```

Nomeação de exceções

Uma exceção é tomada pelo manipulador não pelo seu tipo mas sim por um objeto. Havendo necessidade de transmitir alguma informação do levantamento da exceção para o manipulador, é necessário haver algum mecanismo de colocar tal informação neste objeto. Isto é feito colocando-se campos dentro da classe que representa a exceção e tomando seus valores nos manipuladores. Para tomarmos estes valores nos *handles*, é necessária a definição de um nome para o objeto criado na acusação.
No exemplo criado, é importante saber qual o valor que foi usado como índice na exceção *Vector::Range* (indexação fora dos limites):

```
class Vector{
// ...
        public:
              class Range {
                    public:
                          int index;
// criação do campo que diz o valor inválido
                          Range( int i ) { index = i; }
// a criação do objeto indica o valor inválido
              };
              int& operator[]( int i )
// ...
};
int& Vector::operator[]( int i )
{
        if ( 0<=i && i<sz ) return p[i];
        throw Range(i);
// acusa-se a exceção indicando
// o valor índice invalido ao construir Range
}
```

Para examinar o índice incorreto, o manipulador deve dar um nome ao objeto da exceção:

```
void f( Vector& v )
{ /...
        try {
              qualquer_operação(v);
        }
        catch( Vector::Range r ) {
// 'r' é o nome do objeto Range acusado no operador []
              printf( "índice errado: %d \n", r.index );
              exit(0);
        }
// ...
```

```
};
```

É interessante notar que no caso de templates, tem-se a opção de nomear a exceção de modo que cada classe instanciada pela template tenha sua própria classe de exceção:

```
template<class T> class Allocator {
// ...
        class Exhausted {}; // classe do tratamento de exceção
// ...
};
void f (Allocator<int>& ai, Allocator<double>& ad )
{
        try {
// ...
        }
        catch (Allocator<int>::Exhausted) {
// ...
// tratamento de inteiros
        }
        catch (Allocator<double>::Exhausted) {
// ...
// tratamento de doubles
        }
}
```

Alternativamente, uma exceção pode ser comum a todas as classes instanciadas pela template:

```
class Allocator_Exhausted {};
        template<class T> class Allocator {
// ...
};
void f( Allocator<int>& ai, Allocator<double>& ad )
{
        try {
// ...
        }
        catch( Allocator_Exausted ) {
// ...
// tratamento para ambos os tipos
        }
}
```

Agrupamento de exceções

Normalmente as exceções podem ser categorizadas em famílias. Por exemplo, pode-se imaginar um erro matemático que inclua as exceções de *overflow*, *underflow*, divisão por zero, etc. A exceção de erro matemático (MATHERR) pode ser determinada pelo conjunto de erros que podem ser produzidos em uma biblioteca de funções numéricas. Uma maneira de fazer MATHERR é determiná-la como um tipo de todos os possíveis erros numéricos:

```
enum MATHERR { Overflow, Underflow, ZeroDivide };
```

e na função que trata os erros:

```
void f( .... )
{
        try {
// ...
        }
        catch( MATHERR m ) {
```

```
                switch( m ) {
                      case Overflow:
// ...
                      case Underflow:
// ...
                      case ZeroDivide:
// ...
                }
        }
}
```

De outra maneira, C++ usa a capacidade de herança e de funções virtuais para evitar este tipo de switch. É possível a utilização de herança para descrever coleções de exceções. Por exemplo:

```
class MATHERR {};
class Overflow: public MATHERR {} ;
class Underflow: public MATHERR {} ;
class ZeroDivide: public MATHERR {} ;
// ....
```

Para este caso, existem muitas ocasiões em que deseja-se fazer o tratamento de MATHERR sem saber precisamente de que tipo é o erro. Com a utilização de herança, é possível dizer:

```
try {
// ...
}
catch( Overflow ) {
// tratamento de overflow ou tudo derivado de overflow
}
catch ( MATHERR ) {
// tratamento de qualquer outro erro numérico
}
```

A organização de exceções em hierarquias pode ser importante para a robustez do código de um programa. Consideremos o tratamento de todas as exceções de nossa biblioteca numérica sem o agrupamento destas. Neste caso, as funções que utilizam esta biblioteca teriam que exaustivamente determinar e tratar toda a lista de erros.

```
try {
// ...
}
catch (Overflow) { /* ..... */ }
catch (Underflow) { /* ..... */ }
catch (ZeroDivide) { /* ..... */ }
// e todas as outras exceções!!!
```

Isto não somente é tedioso mas dá margem ao esquecimento de alguma exceção. Além, uma determinada função que desejar fazer o tratamento de qualquer erro numérico (sem saber que tipo de erro) precisa ser constantemente atualizada quando do aparecimento de novas exceções. Por exemplo: o logaritmo de número menor ou igual a zero; o que implica também em recompilação destas funções. Neste sentido é muito mais prático fazer:

```
try {
// ...
}
catch (MATHERR) { /* ..... */ }
// trata qualquer erro matemático.
```

que garante sempre o tratamento sem a necessidade de manutenção do código quando da introdução de novas exceções.

Exceções derivadas

A utilização de hierarquias de exceções naturalmente direciona os manipuladores que estão interessados somente em um subconjunto da informação carregada pelas exceções. Em outras palavras, uma exceção é normalmente tomada por um manipulador da classe básica ao invés de um da classe exata. A semântica para a tomada de um manipulador e nomeação de uma exceção é idêntica para a passagem de argumentos de funções vista anteriormente. Por exemplo:

```
class MATHERR {
// ...
        virtual void debug_print() {};
};
class int_overflow : public MATHERR {
        public:
              char op;
              int opr1, opr2;
              int_overflow( const char p, int a, int b )
              { op=p; opr1=a; opr2=b; }
              virtual void debug_print() // redefinição de debug_print
              { printf(" operador:%c:( %d, %d )", op, opr1, opr2 ); }
              };
              void f()
              {
                    try{
                          g();
                    }
                    catch( MATHERR m ) { /* ... */ }
}
```

Quando um manipulador MATHERR é encontrado, *m* é um objeto MATHERR mesmo que a chamadade *g* tenha acusado um *int_overflow*. Isto implica que a informação extra encontrada em *int_overflow* está inacessível; isto é, se dentro do tratador chamarmos a função *debug_print*, não conseguiremos ver nada sobre o erro de *overflow* de inteiros. Isto é devido ao fato do compilador não fazer *late-binding* com o objeto. No entanto, ponteiros e referências podem ser utilizados para evitar esta perda de informação. Para tal, pode-se escrever:

```
int add( int x, int y )
{
        if ( x>0 && y>0 && x>MAXINT - y|| x<0 && y<0 && x<MININT + y )
              throw int_overflow( '+', x, y );
        return x + y;
}
void f()
{
        try {
              add( 1, 2 ); // ok
              add( MAXINT, 3 ); // causa exceção
        }
        catch( MATHERR& m ) { // recebe no manipulador uma referência
// ...
        m.debug_print();
// chama o método de int_overflow!!!
        }
}
```

Re-throw

Dada uma função que capture uma exceção, não é incomum para um manipulador chegar a conclusão que nada pode ser feito a respeito do erro. Neste caso, a coisa típica a ser feita é a acusação da exceção novamente (*re-throw*), esperando que outro manipulador possa fazê-lo melhor. Por exemplo:

```
void h()
{
        try {
// ...
        }
        catch( MATHERR ) {
              if ( posso_tratar() ) tratamento();
              else throw; // re-throw
        }
}
```

Um *re-throw* é indicado pelo comando throw sem argumentos. A exceção de relevantamento é a exceção original tomada e não somente a parte que era acessível como MATHERR. Em outras palavras, se um *int_overflow* foi acusado, a função que chamou h pode ainda tomar um int_overflow que h tomou como MATHERR e decidiu reacusar.

```
void k()
{
        try {
              h();
              // ...
        }
        catch( int_overflow ) {
              // ...
        }
}
```

A versão abaixo deste tipo de comportamento pode ser útil. Assim como em funções, pode-se utilizar '...' (indicando qualquer argumento) de modo que catch(...) signifique qualquer exceção. Por exemplo:

```
void m()
{
        try {
              // ...
        }
        catch(...) {
              limpeza();
              throw;
        }
}
```

Isto é, se qualquer exceção ocorrer, resultado da execução de parte de *m()*, a função *limpeza()* será chamada no manipulador e a exceção que causou a chamada da função *limpeza()* será reacusada. Devido ao fato de que exceções derivadas podem ser tratadas por manipuladores para mais de um tipo de exceção, a ordem em que os estes aparecem após o bloco de *try* é relevante. Os tratadores são escolhidos em ordem. Por exemplo:

```
        try {
              // ...
        }
```

```
catch( ibuf ) {
      // tratador de input overflow
}
catch( io ) {
      // tratador de qualquer erro de I/O
}
catch( stdlib ) {
      // tratador de qualquer erro em bibliotecas
}
catch( ... ) {
      // tratador de qualquer outra exceção
}
```

Especificação de exceções

A acusação e o tratamento de exceções afetam o relacionamento entre as funções. Neste sentido, é interessante haver um mecanismo de especificar quais exceções que podem ser levantadas como parte da declaração de uma função. Por exemplo:

void f(int a) throw (x2, x3, x4);

especifica que a função *f* só pode acusar as exceções *x2, x3, x4* e suas derivadas, nada mais. Deste modo, está garantindo para quem a chama que durante sua execução nenhuma outra exceção será levantada. Se, por acaso, algo acontecer que invalide esta garantia, a tentativa de acusação de uma exceção indevida será transformada em uma chamada para a função *unexpected*. O significado default para *unexpected* é a chamada a terminate, que normalmente representa um *abort*.
Desta forma, escrever:

```
void f( int a ) throw (x2, x3, x4)
{ /* implementacao qualquer */ }
```

significa a mesma coisa que:

```
void f( int a )
{
        try{
              /* implementação qualquer */
        }
        catch(x2) { throw; } // re-throw
        catch(x3) { throw; } // re-throw
        catch(x4) { throw; } // re-throw
        catch(...){ unexpected(); }
}
```

Mais que economia de digitação, o uso de especificação de exceções explicita as exceções que podem surgir na definição da função (.h) o que nem sempre aconteceria se esta definição ficasse em sua implementação. Uma função sem especificação pode levantar qualquer exceção.

int f ();

enquanto que uma função sem a possibilidade de acusar qualquer exceção é declarada com uma lista explicitamente vazia:

int g() throw();

Exceções indesejadas

O mau uso de especificações de exceções pode levar a chamadas a função *unexpected*, que é indesejável a não ser no caso de testes. Pode-se evitar isto por uma boa estruturação e organização das exceções ou pela interceptação das chamadas a *unexpected*.
A função *set_unexpected* serve para interceptarmos estes casos. Esta função redefine o comportamento do sistema quando de uma exceção indesejada retornando o tratador antigo.

Abaixo temos um exemplo deste mecanismo. Neste caso, cria-se uma classe que representa um trecho aonde exceções não previstas devem ser tratadas.

```
typedef void(*functype)();
functype set_unexpected( functype );
class MyPart {
        functype old;
        public:
              MyPart( functype f ) { old = set_unexpected( f ); }
              ~MyPart() { set_unexpected( old ); }
}
void new_trat() { printf("novo tratamento.\n"); }
void f()
{
        MyPart( &new_trat ); // construtor implica em redefinição
        g();
} // destrutor reseta unexpected() anterior
```

Neste caso, a execução de f é protegida contra erros de exceções não desejadas.

Exceções não tratadas

Uma exceção acusada e não tratada implica na chamada da função *terminate*. Esta também é chamada se o mecanismo de exceções de C++ encontrar a pilha corrompida. terminate executa a última função recebida como argumento da função *set_terminate*.

Este mecanismo serve como mais um nível para erros de exceção. É normalmente utilizado para medidas mais drásticas no sistema como: aborto da execução do processo, reinicialização do sistema, etc. A redefinição deste comportamento é feita de modo análogo ao *unexpected*.

```
typedef void (*PFV) ();
PFV set_terminate (PFV);
```