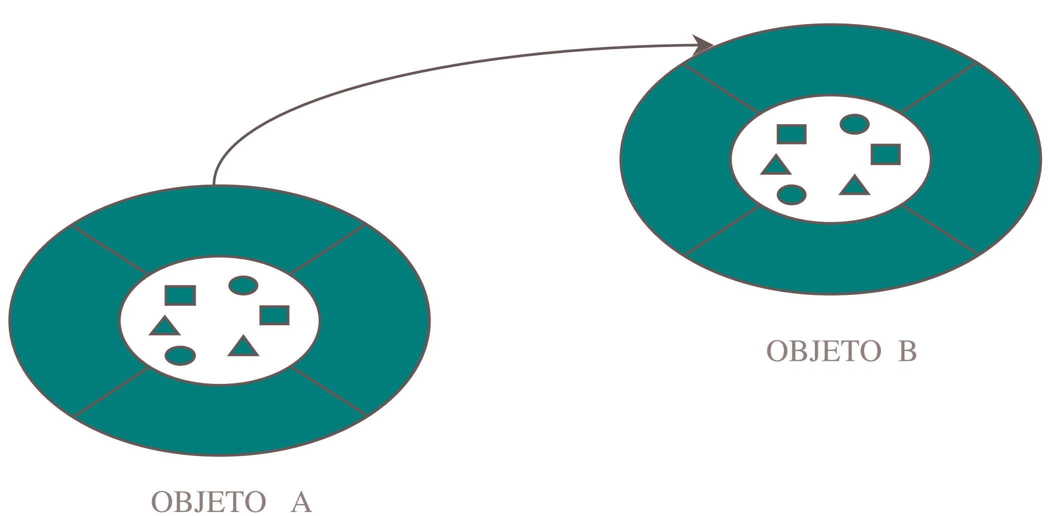
Programação Orientada a Objetos



Rafael Rodrigues de Souza

Sumário

Introdução 3
Programação estruturada 3
Programação orientada a objeto 3
Classes 4
Encapsulamento 4
Atributos (Propriedades/Variáveis) 5
Métodos (Serviços/Funções) 5
Herança (Hereditariedade) 6
Polimorfismo 6
Programação Orientada a Objetos usando C++ 7
Classes e Objetos 7
Especificando uma Classe 8
Struct em C++ 9
Atributos ou dados membro 9
Métodos ou Funções Membro. 10
Funções membro que retornam valores 13
Construtores 14
Encapsulamento com Class 18
Atributos Private, Funções membro Public 20
Um dado membro é Public 21
Compilando um Programa com Vários Arquivos 22
Considerações C++: 24
Const 24
Funções Inline 25
Vetores criados estaticamente 28
Copia de Objetos com Vetores Alocados Estaticamente 28
Vetores criados dinamicamente 29
Copia de Objetos com Vetores Alocados Dinamicamente 30
Referência & 31
Argumentos de Linha de Comando 32
Herança 34
Hierarquias de tipos 34
Herança Pública e Privada 41
Herança Múltipla 41
Funções Amigas 42
Sobrecarga de Operadores 44
Sobrecarga de Operadores Unários 44
Sobrecarga de Operadores Binários 45
Funções Virtuais 46
Funções Virtuais Puras 47
Gabaritos de funções 48
Gabarito de classes 48
Referências Bibliográficas 50

Introdução

As primeiras linguagens de programação eram bastante rústicas e obrigavam o programador a conhecer em excesso as características do hardware que estava usando. Um programa se dirigia para um equipamento especifico e era extremamente complexo de desenvolver. Os programas eram desenvolvidos em linguagens de baixo nível como o assembler. Com o passar dos anos, desenvolveram-se novas linguagens de programação, que iam desvinculando o programa do hardware.
Enquanto o desenvolvimento de hardware se dava a passos largos, o desenvolvimento de softwares estava atrasado cerca de 20 anos.

As linguagens de programação mais modernas permitem que um programa seja compilado e rodado em diferentes plataformas.
Mesmo com o surgimento de novas linguagens de programação, as equipes de programação sempre tiveram enormes problemas para o desenvolvimento de seus programas. Tendo sempre que partir do zero para o desenvolvimento de um novo programa, ou reaproveitando muito pouco dos códigos já desenvolvidos.

Programação estruturada

Com o desenvolvimento das técnicas estruturadas, os problemas diminuíram.
Na programação estruturada as funções trabalham sobre os dados, mas não tem uma ligação íntima com eles.

Programação orientada a objeto

Para tentar solucionar o problema do baixo reaproveitamento de código, tomou corpo a ideia da Programação Orientada a Objeto (POO). A POO não é nova, sua formulação inicial data de 1960. Porém, somente a partir dos anos 90 é que passou a ser usada.

Hoje, todas as grandes empresas de desenvolvimento de programas tem desenvolvido os seus software's usando a programação orientada a objeto.
A programação orientada a objeto difere da programação estruturada. Na programação orientada a objeto, funções e dados estão juntos, formando o objeto.

Esta abordagem cria uma nova forma de analisar, projetar e desenvolver programas de uma forma mais abstrata e genérica, que permite um maior reaproveitamento dos códigos e facilita a manutenção. A programação orientada a objeto não é somente uma nova forma de programar é uma nova forma de pensar um problema, de forma abstrata, utilizando conceitos do mundo real e não conceitos computacionais. Os conceitos de objetos devem acompanhar todo o ciclo de desenvolvimento de um software.

A programação orientada a objeto também incluí uma nova notação e exige pôr parte do analista/programador o conhecimento desta notação (diagramas],

Classes

Quando falamos de classes, lembramos de classes sociais, de classes de animais (os vertebrados), de classes de objeíos da natureza, de hierarquias. Ou seja, uma classe descreve um grupo de objetos com os mesmo atributos e comportamentos, alem dos mesmos relacionamentos com outros objetos.
Para a análise orientada a objeto, uma classe è um conjunto de códigos de programação que incluem a definição dos atributos e dos métodos necessários para a criação de um ou mais objetos.
A classe contém toda a descrição da forma do objeto, é um molde para a criação do objeto, é uma matriz geradora de objetos, è uma fábrica de objetos- Uma classe também é um tipo definido pelo usuário.

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

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

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

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

Encapsulamento

Todos os equipamentos que utilizamos são altamente encapsulados, Tome como exemplo a sua televisão, ela tem um pequeno conjunto de botões que lhe permitem manipular os atributos do objeto televisor que são de seu interesse, como o canal, o volume, as cores.
Mas você sabe que o funcionamento do objeto televisor é extremamente complexo e que ao selecionar um novo canal, uma série de atributos internos são processados e alterados. Os atributos e funções internas estão encapsuladas, escondidas de você.
Para a analise orientada a objeto, encapsulamento é o ato de esconder do usuário informações que não são de seu interesse. O objeto atua como uma caixa preta, que realiza determinada operação mas o usuário não sabe, e não precisa saber,

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

Exemplos:

• Um computador é um objeto extremamente complexo, mas para o usuário o que importa é o teclado, o monitor, o mouse e o gabinete.

•Ao utilizar um software como o StarOffice, a forma de uso é a mesma, seja num Pentium 4 ou num AMD K6, Os elementos invisíveis do computador (placa mãe, processador, memória) não alteram a forma de usar programa.

Atributos (Propriedades/Variáveis)

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

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

Exemplo: Um contador de relógios criados.

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

Exemplo: A hora de um relógio,
Cada. relógio tem uma hora, que pode ou não estar certa.

Métodos (Serviços/Funções)

A todo objeto podemos relacionar determinados comportamentos, ações e reações. As ações ou comportamento dos objetos são chamadas na análise orientada a objeto de métodos, assim, um método *é* uma função, um serviço fornecido pelo objeto.
Os comportamentos dos objetos são definidos na classe através dos métodos e servem para manipular e alterar os atributos do objeto (alteram o estado do objeto).

Exemplos:

• Um automóvel tem o comportamento de se locomover.

• Um computador de processar programas.

• Uma. edificação de dar abrigo.

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

Herança (Hereditariedade)

A herança esta relacionada as hierarquias e as relações entre os objetos. No dia a dia, quando se fala de herança se refere a transferência de propriedades de um pai aos seus filhos, ou seja, aquilo que é do pai passa a ser do filho.
E comum ainda o dito popular "puxou o pai'*, que significa que o filho tem as mesmas características do pai. De uma maneira geral as pessoas sabem que o filho puxou o pai mas não é ele, ou seja não são a mesma pessoa. E que o filho apresenta determinadas características diferentes de seu pai.

Na análise orientada a objeto, herança é o mecanismo em que uma classe filha compartilha automaticamente todos os métodos e atributos de sua classe pai. A herança permite implementar classes descendentes implementando os métodos e atributos que se diferenciam da classe pai.

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

Exemplo:

• Um Pentium II tem todas as características do Pentium preservadas, mas acrescentou mais memória cache, a memória cache já existia mas foi ampliada.

• Uma placa mãe nova apresenta a interface USB, é uma novidade que antes não existia.

Polimorfismo

A palavra polimorfismo significa muitas formas, e representa o fato de uma determinada característica (potência do motor do veiculo) ser diferente para cada

filho (tipo de veiculo). Na natureza o conceito de polimorfismo é inerente ao processo de desenvolvimento, os seres evoluem, se modificam.

Em suma, estamos partindo de um objeto mais simples e evoluindo. Mas os conceitos do objeto pai continuam a existir nos objetos descendentes, mesmo que tenham sofrido modificações, aperfeiçoamentos e assumido novas formas(polimorfismo).
O conceito de polimorfismo é fundamental para a análise orientada a objeto e sua aplicação se fundamenta no uso de uma superclasse, através do qual vamos desenvolver nossa hierarquia de classes

Programação Orientada a Objetos usando C++

Classes e Objetos

Uma classe é um tipo definido pelo usuário que contém o molde, a especificação para os objetos, assim como o tipo inteiro contém o molde para as variáveis declaradas como inteiros. A classe envolve, associa, funções e dados, controlando o acesso a estes, definí-la implica em especificar os seus atributos (dados) e suas funções membro (código).

Um programa que utiliza uma interface controladora de um motor elétrico provavelmente definiria a classe motor. Os atributos desta classe seriam: temperatura, velocidade, tensão aplicada. Estes provavelmente seriam representados na classe por tipos como float ou long . As funções membro desta classe seriam funções para alterar a velocidade, ler a temperatura, etc.

Um programa editor de textos definiria a classe parágrafo que teria como um de seus atributos uma string ou um vetor de strings, e como funções membro, funções que operam sobre estas strings. Quando um novo parágrafo é digitado no texto, o editor cria a partir da classe parágrafo um objeto contendo as informações particulares do novo texto. Isto se chama instanciação ou criação do objeto.

Classes podem ser declaradas usando a palavra reservada struct ou a palavra reservada class, nos exemplos posteriores entraremos em mais detalhes. As classes do próximo tópico são declaradas com struct por razões didáticas. Quando chegarmos em encapsulamento mostraremos como declarar classes com class e não usaremos mais struct no tutorial.

Objetos são instâncias de uma classe. Quando um objeto é criado ele precisa ser inicializado, ou seja para uma única classe : Estudante de graduação podemos ter vários objetos num programa: Estudante de graduação Carlos, Identificação 941218, Curso Computação; Estudante de graduação Luiza , Identificação 943249, Curso Engenharia Civil... A classe representa somente o molde para a criação dos objetos, estes sim contém informação, veja tópico classes e objetos.

Especificando uma Classe

Suponha um programa que controla um motor elétrico através de uma saída serial. A velocidade do motor é proporcional a tensão aplicada, e esta proporcional aos bits que vão para saída serial e passando por um conversor digital analógico.

Vamos abstrair todos estes detalhes por enquanto e modelar somente a interface do motor como uma classe, a pergunta é que funções e que dados membro deve ter nossa classe, e que argumentos e valores de retorno devem ter essas funções membro:

Representação da velocidade:

A velocidade do motor será representada por um atributo, ou dado membro, inteiro (int). Usaremos a faixa de bits que precisarmos, caso o valor de bits necessário não possa ser fornecido pelo tipo , usaremos então o tipo long , isto depende do conversor digital analógico utilizado e do compilador.

Representação da saída serial:

O motor precisa conhecer a sua saída serial, a sua ligação com o "motor do mundo real". Suponha uma representação em hexadecimal do atributo endereço de porta serial, um possível nome para o atributo: enderecomotor. Não se preocupe em saber como usar a representação hexadecimal.

Alteração do valor da velocidade:

Internamente o usuário da classe motor pode desejar alterar a velocidade, cria-se então o método (em C++ função membro): void altera_velocidade(int novav); . O código anterior corresponde ao cabeçalho da função membro, ela é definida junto com a classe motor, associada a ela. O valor de retorno da função é void (valor vazio), poderia ser criado um valor de retorno (int) que indicasse se o valor de velocidade era permitido e foi alterado ou não era permitido e portanto não foi alterado.

Não faz sentido usar, chamar, esta função membro separada de uma variável do tipo motor, mas então porque na lista de argumentos não se encontra um motor? Este pensamento reflete a maneira de associar dados e código (funções) das linguagens procedurais. Em linguagens orientadas a objetos o código e os dados são ligados de forma diferente, a própria declaração de um tipo definido pelo usuário já engloba as declarações das funções inerentes a este tipo.

Note que não fornecemos o código da função, isto não é importante, por hora a preocupação é com a interface definida pela classe: suas funções membro e dados membro. Apenas pense que sua interface deve ser flexível de modo a não apresentar entraves para a criação do código que seria feita numa outra etapa. Nesta etapa teríamos que imaginar que o valor numérico da velocidade deve ir para o conversor onde irá se transformar numa diferença de potencial a ser aplicada nos terminais do motor, etc.

Um diagrama simplificado da classe motor com os dados membro e as funções membro:

Struct em C++

Objetos são instâncias de uma classe. Quando um objeto é criado ele precisa ser inicializado, ou seja para uma única classe : Estudante de graduação podemos ter vários objetos num programa: Estudante de graduação Carlos, Identificação 941218, Curso Computação; Estudante de graduação Luiza , Identificação 943249, Curso Engenharia Civil... A classe representa somente o molde para a criação dos objetos, estes sim contém informação, veja tópico classes e objetos.

Atributos ou dados membro

Este exemplo declara uma struct e em seguida cria um objeto deste tipo em main alterando o conteúdo desta variável. Uma struct é parecida com um record de Pascal, a nossa representa um círculo com os atributos raio, posição x , posição y, que são coordenadas cartesianas. Note que este objeto não possui funções membro ainda.

```
#include <iostream.h>
struct circulo //struct que representa um circulo.

{float raio;
 float x; //posicoes em coordenadas cartesianas
 float y; };

void main()
 {circulo ac; //criacao de variavel , veja comentarios.
  ac.raio=10.0; //modificacao de conteudo (atributos) da struct
  ac.x=1.0; //colocando o circulo em uma posicao determinada
  ac.y=1.0; //colocando o circulo em uma posicao determinada
  cout << "Raio:"<<ac.raio <<endl; //verificacao dos atributos alterados.
  cout << "X:"<<ac.x << "\n"; // "\n"==endl
  cout << "Y:" <<ac.y<< endl;}
```

Resultado do programa:

Raio:10
X:1
Y:1

Comentários:

```
struct circulo //struct que representa um circulo.
{
 float raio;
 float x; //posicoes em coordenadas cartesianas
 float y;
};
```

Este código é a declaração da classe círculo, entre chaves vem os dados membro e as funções membro que não foram apresentadas ainda.

A sintaxe para criação de objetos da classe círculo (circulo ac;) , por enquanto não difere da sintaxe para a criação de variáveis do tipo int.

O acesso aos dados membro deve ser feito usando o nome do objeto e o nome do dado membro, separados por um ponto: ac.raio=10.0; . Note que raio sozinho não faz sentido no programa, precisa-se especificar de que objeto se deseja acessar o raio.

Aos que programam em C:

Os programadores C podem notar algo interessante: "C++ não requer a palavra struct na declaração da variável, ela se comporta como um tipo qualquer: int , float ...". Outros programadores que não haviam usado struct previamente em C não se preocupem, façam apenas os exercícios deste exemplo e estarão aptos a prosseguir.

Exercícios:

1) Repita o mesmo exemplo só que agora mova o círculo alterando as componentes x e y. Ponha o círculo em (0.0,0.0) através de atribuições do tipo ac.x=1.0; mova o círculo para (1.0,1.0). Acompanhe todas as modificações da struct através de cout's.
2)Simplifique o programa anterior retirando o atributo raio. Você pode dar o nome de ponto ou ponto_geometico para esta classe.

Métodos ou Funções Membro.

C++ permite que se acrescente funções de manipulação da struct em sua declaração, juntando tudo numa só entidade que é uma classe. Essas funções membro podem ter sua declaração (cabeçalho) e implementação (código) dentro da struct ou só o cabeçalho (assinatura) na struct e a implementação, código, fora. Este exemplo apresenta a primeira versão, o próximo a segunda versão (implementação fora da classe).

Essas funções compõem a interface da classe. A terminologia usada para designá-las é bastante variada: funções membro, métodos, etc. Quando uma função membro é chamada, se diz que o objeto está recebendo uma mensagem (para executar uma ação).

Um programa simples para testes sobre funções membro seria o seguinte:

```
#include <iostream.h>

struct contador //conta ocorrencias de algo
{
 int num; //numero do contador
 void incrementa(void){num=num+1;}; //incrementa contador
 void comeca(void){num=0;}; //comeca a contar
};
 void main() //teste do contador
{
```

```
 contador umcontador;
 umcontador.comeca(); //nao esqueca dos parenteses, e uma funcao membro e nao
 atributo!
 cout << umcontador.num << endl;
 umcontador.incrementa();
 cout << umcontador.num << endl;
}
```

Resultado do programa:

0
1

Comentários:

O programa define um objeto que serve como contador, a implementação representa a contagem no atributo num que é um número inteiro. As funções membro são simples: incrementa adiciona um ao contador em qualquer estado e inicializa a contagem em zero.

Sintaxe:

A sintaxe para declaração de funções membro dentro de uma classe é a mesma sintaxe de declaração de funções comuns : tipoderetorno nomedafuncao(lista_de_argumentos) { /*codigo */ }. A diferença é que como a função membro está definida na classe, ela ganho acesso direto aos dados membros, sem precisar usar o "ponto", exemplo um_objeto.dadomembro; . Lembre-se que as chamadas de funções membro já se referem a um objeto específico, embora elas sejam definidas de uma forma geral para toda a classe.

A sintaxe de chamada ou acesso à funções membro é semelhante a sintaxe de acesso aos dados membro com exceção dos parênteses que contém a lista de argumentos da função, mesmo que a lista seja vazia eles devem estar presentes: umcontador.incrementa();. Primeiro insere-se o nome do objeto e depois a chamada da função, estes são separados por um ponto. Cuidado para não esquecer os parênteses nas chamadas de funções membro em programas futuros, este é um erro bastante comum.

Agora o programa mais complicado:

```
#include <iostream.h> //para cout

struct circulo
{
 float raio;
 float x; //atributo coordenada cartesiana x
 float y; //atributo coordenada cartesiana y

 void move(float dx,float dy) //função membro ou função membro move
 {
  x+=dx; //equivale a x=x+dx;
  y+=dy;
 }

 void mostra(void) //função membro ou função membro mostra
 {
```

```
 cout << "Raio:"<<raio <<endl;
 cout << "X:"<<x << endl;
 cout << "Y:" <<y<< endl;}
};
void main()
{
 circulo ac;// * instanciação de um objeto circulo (criacao)
 ac.x=0.0;
 ac.y=0.0;
 ac.raio=10.0;
 ac.mostra();
 ac.move(1.0,1.0);
 ac.mostra();
 ac.x=100.0;
 ac.mostra();
}
```

Resultado do programa:

Raio:10
X:0
Y:0
Raio:10
X:1
Y:1
Raio:10
X:100
Y:1

Comentários:

A função membro move altera as coordenadas do objeto. O objeto tem suas coordenadas x e y somadas com os argumentos dessa função membro. Note que esta função membro representa uma maneira mais segura, clara, elegante de alterar as coordenadas do objeto do que acessá-las diretamente da seguinte forma: `ac.x+=dx;`. `ac.y+=dy;`. Lembre-se que `ac.x+=dx` é uma abreviação para `ac.x=ac.x+dx;` .

Como funcionam no compilador as chamadas de funções membro:

É possível imaginar que as definições de funções membro ocupam um grande espaço na representação interna dos objetos, mas lembre-se que elas são todas iguais para uma classe então basta manter para cada classe uma tabela de funções membro que é consultada no momento da chamada . Os objetos só precisam ter uma referência para esta tabela.

Exercícios:

1)Neste mesmo programa, crie uma função para a struct chamada "inicializa" que deve ter como argumentos um valor para x, um para y e outro para o raio, e deve alterar os atributos inicializando-os com os valores passados.
Você pode abstrair o uso dessa função como uma maneira de inicializar o objeto de uma só vez embora a função o faça seqüencialmente. Comente as vantagens de fazê-lo, comparando com as outras opções, tenha sempre em mente a questão de segurança quando avaliar técnicas diferentes de programação.

2)No programa anterior, verifique que nada impede que você acesse diretamente os valores de x , y e raio e os modifique. Como você pode criar um número enorme de funções : altera_x(float a); move_raio(float dr); seria desejável que somente essas funções pudessem modificar x, y e raio. Você verá que isso é possível em encapsulamento. Por hora, crie essas funções.

3)Teste a função membro move com argumentos negativos, exemplo ac.move(-1.0,-1.5);. O resultado é coerente?

4)Crie uma nova struct que representa um ponto, que informações você precisa armazenar? Que funções seriam úteis ? Faça um programa para testar sua classe.

5)Melhore a classe contador, defina uma função que imprime o contador na tela. Se você estivesse fazendo um programa para rodar numa interface gráfica com o usuário esta função de imprimir na tela seria a mesma? Definir uma função que retorna uma copia do valor atual do contador garante maior portabilidade? Por quê?

6)"Há uma tendência em definir o maior número de funções membro em uma classe, porque nunca se pode prever exatamente o seu uso em programas futuros". Comente esta frase, tendo em vista o conceito de portabilidade. Você já é capaz de citar outras medidas que tornem suas classes mais portáveis? Leia o exercício anterior.

Funções membro que retornam valores

Até agora só tínhamos visto funções membro com valor de retorno igual a void. Uma função membro, assim como uma função comum, pode retornar qualquer tipo, inclusive os definidos pelo usuário. Sendo assim, sua chamada no programa se aplica a qualquer lugar onde se espera um tipo igual ou equivalente ao tipo do seu valor de retorno, seja numa lista de argumentos de outra função , numa atribuição ou num operador como o cout << variavel;

```
#include <iostream.h>
struct contador //conta ocorrencias de algo
{
  int num; //numero, posicao do contador
```

```
  void incrementa(void){num=num+1;
 }; //incrementa contador

void comeca(void)
{
 num=0;
};//comeca a contar, "reset"

int retorna_num(void) {return num;};};

void main() //teste do contador
{
 contador umcontador;
 umcontador.comeca(); //nao esqueca dos parenteses, e uma funcao membro não dado!
 cout << umcontador.retorna_num() << endl;
 umcontador.incrementa();
 cout << umcontador.retorna_num() << endl;
}
```

Resultado do programa:

0
1

Construtores

Construtores são funções membro especiais chamadas pelo sistema no momento da criação de um objeto. Elas não possuem valor de retorno, porque você não pode chamar um construtor para um objeto. Construtores representam uma oportunidade de inicializar de forma organizada os objetos, imagine se você esquece de inicializar corretamente ou o faz duas vezes, etc.

Um construtor tem sempre o mesmo nome da classe e não pode ser chamado pelo usuário desta. Para uma classe string o construtor teria a forma `string(char* a);` com o argumento `char*` especificado pelo programador. Ele seria chamado automaticamente no momento da criação, declaração de uma string:

```
string a("Texto"); //alocacao estática implica na chamada do construtor
a.mostra(); //chamada de metodos estatica.
```

Existem variações sobre o tema que veremos mais tarde: Sobrecarga de construtor, "copy constructor", como conseguir construtores virtuais (avançado, não apresentado neste texto) , construtor de corpo vazio.

O exemplo a seguir é simples, semelhante aos anteriores, preste atenção na função membro com o mesmo nome que a classe `(struct)` , este é o construtor:

```
#include <iostream.h>

struct ponto
{
 float x;
 float y;
 public:
 ponto(float a,float b);//esse e o construtor, note a ausencia do valor de retorno
```

```
 void mostra(void);
 void move(float dx,float dy);
};

 ponto::ponto(float a,float b) //construtor tem sempre o nome da classe.
 {
  x=a; //incializando atributos da classe
  y=b; //colocando a casa em ordem
 }

 void ponto::mostra(void)
 {
  cout << "X:" << x << " , Y:" << y << endl;
 }

 void ponto::move(float dx,float dy)
 {
  x+=dx;
  y+=dy;
 }

 void main()
 {
  ponto ap(0.0,0.0);
  ap.mostra();
  ap.move(1.0,1.0);
  ap.mostra();
 }
```

Resultado do programa:

X:0 , Y:0
X:1 , Y:1

Comentários:

Deixaremos para nos aprofundarmos em alocação dinâmica e construtores. Por hora você pode se ater ao uso estático de objetos.

Note que com a definição do construtor, você é obrigado a passar os argumentos deste no momento da criação do objeto.

Exercícios:

1)Você sabia que uma função membro pode chamar outra função membro? Parta do exemplo de e crie um construtor que chama a antiga função inicializa(float a,float b) passando os argumentos do construtor:

```
ponto(float a, float b)
{inicializa(a,b); }
```

Na chamada de inicializa() fica implícito que ela se aplica ao objeto cujo construtor foi chamado. Isto é válido também para outras funções membro que

chamam funções membro, ou seja, não é necessário o operador identificador.inicializa(a,b).

Este exercício é útil para mostrar outro recurso de C++, o ponteiro this. This é uma palavra reservada e dentro de qualquer função membro this é um ponteiro para o objeto em questão, então o código descrito acima poderia também assumir a seguinte forma equivalente:

```
ponto(float a, float b)
{this->inicializa(a,b); } //Verifique!
```

É lógico que neste exemplo this não tem muita utilidade, mas existem casos onde um objeto precisa passar seu ponteiro para alguma função que o modifica, ou fica possuindo uma referência para ele, então usa-se this. Veremos outras aplicações mais adiante.

2) Introduza mensagens no construtor tipo: `cout << "Objeto instanciado.";` introduza também trechos no seu programa parecidos com: `cin >> a;` , só para que o programa não rode todo de uma vez, rapidamente. O objetivo é acompanhar visualmente a seqüência de criação e modificação dos objetos.

3)Crie uma classe reta que tem como atributos dois objetos da classe ponto.

Dica: **Não use construtores**, use funções do tipo inicializa(), já apresentadas. Quando seu programa ficar pronto acrescente função membros para esta reta tais como inclinação, coeficiente linear, etc. Existem maneiras mais eficientes e compactas de representar uma reta, porém faça como foi sugerido, neste caso o objetivo não é eficiência.

Destrutores

Análogos aos construtores, os destrutores também são funções membro chamadas pelo sistema, só que elas são chamadas quando o objeto sai de escopo ou em alocação dinâmica, tem seu ponteiro desalocado, ambas (construtor e destrutor) não possuem valor de retorno.

Você não pode chamar o destrutor, o que você faz é fornecer ao compilador o código a ser executado quando o objeto é destruído, apagado. Ao contrário dos construtores, os destrutores não tem argumentos.

Os destrutores são muito úteis para "limpar a casa" quando um objeto deixa de ser usado, no escopo de uma função em que foi criado, ou mesmo num bloco de código. Quando usados em conjunto com alocação dinâmica eles fornecem uma maneira muito prática e segura de organizar o uso do "heap". A importância dos destrutores em C++ é aumentada pela ausência de "garbage collection" ou coleta automática de lixo.

A sintaxe do destrutor é simples, ele também tem o mesmo nome da classe só que precedido por ~ , ele não possui valor de retorno e seu argumento é void sempre:

```
~nomedaclasse(void) { /* Codigo do destrutor */ }
```

O exemplo a seguir é simples, porque melhorias e extensões sobre o tema destrutores serão apresentadas ao longo do texto:

```
//destrutor de uma classe
#include <iostream.h>

struct contador
{
 int num;
 contador(int n)
 {
  num=n;
 } //construtor

 void incrementa(void)
 {
  num+=1;
 }//funcao membro comum, pode ser chamada pelo usuario

 ~contador(void)
 {
  cout << "Contador destruido, valor:" << num <<endl;
 } //destrutor
};

void main()
{
 contador minutos(0);
 minutos.incrementa();
 cout << minutos.num << endl;

 {//inicio de novo bloco de codigo
  contador segundos(10);
  segundos.incrementa();
  cout << segundos.num <<endl; //fim de novo bloco de código
 }
 minutos.incrementa();
}
```

Resultado do programa:

1
11
Contador destruido, valor:11
Contador destruido, valor:2

Comentários:

No escopo de main é criado o contador minutos com valor inicial==0.
Minutos é incrementado, agora minutos.num==1.
O valor de num em minutos é impresso na tela.
Um novo bloco de código é criado.

Segundos é criado, instanciado como uma variável deste bloco de código, o valor inicial de segundos é 10, para não confundir com o objeto já criado.
Segundos é incrementado atingindo o valor 11.
O valor de segundos é impresso na tela.
Finalizamos o bloco de código em que foi criado segundos, agora ele sai de escopo, é apagado, mas antes o sistema chama automaticamente o destrutor.
Voltando ao bloco de código de main(), minutos é novamente incrementado.
Finalizamos main(), agora, todas as variáveis declaradas em main()saem de escopo, mas antes o sistema chama os destrutores daquelas que os possuem.

Encapsulamento com Class

Encapsulamento, "data hiding". Neste tópico vamos falar das maneiras de restringir o acesso as declarações de uma classe, isto é feito em C++ através do uso das palavras reservadas public, private e protected. Friends também restringe o acesso a uma classe.

Não mais apresentaremos exemplos de classes declaradas com struct. Tudo que foi feito até agora pode ser feito com a palavra class ao envés de struct, incluindo pequenas modificações. Mas porque usar só class nesse tutorial? A diferença é que os dados membro e funções membro de uma struct são acessíveis por "default" fora da struct enquanto que os atributos e métodos de uma classe não são, acessíveis fora dela (main) por "default". Você nem deve ter se preocupado com isso porque usando struct da forma como usávamos, tudo ficava acessível.

Então como controlar o acesso de atributos e métodos em uma classe? Simples, através das palavras reservadas private, public e protected.

Protected será explicada em herança pois está relacionada a mesma, por hora vamos focalizar nossa atenção em private e public que qualificam os dados membro e funções membro de uma classe quanto ao tipo de acesso (onde eles são visíveis) . Public, private e protected podem ser vistos como qualificadores, "specifiers".

Para facilitar a explicação suponha a seguintes declarações **equivalentes** de classes:

1)
```
class ponto
{
  float x; //dados membro
  float y;
  public: //qualificador
  void inicializa(float a, float b) {x=a; y=b;}; //funcao membro
  void move(float dx, float dy) {x+=dx; y+=dy; }; };
```

a declaração 1 equivale totalmente à:

2)
```
class ponto {

private:
float x;
float y;
```

```
public:
//qualificador
void inicializa(float a, float b)
{
x=a;
y=b;
};

 void move(float dx, float dy)
 {
  x+=dx;
  y+=dy;
 };
};
```

que equivale totalmente à:

3)

```
struct ponto {
private: //se eu nao colocar private eu perco o encapsulamento em struct.
float x;
float y;

public:
//qualificador
 void inicializa(float a, float b)
 {
  x=a; y=b;
 };
 void move(float dx, float dy)
 {
  x+=dx;
  y+=dy;
 };
```

Fica fácil entender essas declarações se você pensar no seguinte: esses qualificadores se aplicam aos métodos e atributos que vem após eles, se houver então um outro qualificador, teremos agora um novo tipo de acesso para os métodos declarados posteriormente.

Mas então porque as declarações são equivalentes? É porque o qualificador private é "default" para class, ou seja se você não especificar nada , até que se insira um qualificador, tudo o que for declarado numa classe é private. Já em struct, o que é default é o qualificador public.

Agora vamos entender o que é private e o que é public:

Vamos supor que você instanciou (criou) um objeto do tipo ponto em seu programa:

ponto meu; //instanciacao

Segundo o uso de qualquer uma das definições da classe ponto dadas acima você **não** pode escrever no seu programa:

meu.x=5.0; //**erro !**, como fazíamos nos exemplos de anteriores, a não ser que x fosse declarado depois de public na definição da classe o que não ocorre aqui. Mas você pode escrever x=5.0; na implementação (dentro) de um método porque

enquanto não for feito uso de herança, porque uma função membro tem acesso a tudo que é de sua classe, veja o programa seguinte.

Você pode escrever: meu.move(5.0,5.0); ,porque sua declaração (move) está na parte public da classe. Aqui o leitor já percebe que podem existir funções membro private também, e estas só são acessíveis dentro do código da classe (outras funções membro).

Visibilidade das declarações de uma classe, fora dela e de sua hierarquia. Veja que só a parte public é visível neste caso:

Visibilidade das declarações de uma classe, dentro dela mesma:

Exercícios:

1) Qual das seguintes declarações permite que se acesse em main **somente** os métodos move e inicializa, encapsulando todos os outros elementos da classe? Obs.: A ordem das áreas private e public pode estar invertida com relação aos exemplos anteriores.

a)
```
struct ponto {//se eu nao colocar private eu perco o encapsulamento em struct.
float x;
float y;
public://qualificador
void inicializa(float a, float b) {x=a; y=b;};
void move(float dx, float dy) ; {x+=dx; y+=dy; }};
```
b)
```
class ponto {
public: //qualificador
void inicializa(float a, float b) {x=a; y=b;};

void move(float dx, float dy) ; {x+=dx; y+=dy; };
private:
float x;
float y;};
```
c)
```
class ponto {
public: //qualificador
void inicializa(float a, float b) {x=a; y=b;}; //funcao membro
void move(float dx, float dy) ; {x+=dx; y+=dy; };
float x;//dados membro
float y;};
```

Atributos Private, Funções membro Public

Aplicando encapsulamento a classe ponto definida anteriormente.

```
#include <iostream.h>
class ponto
{ private: //nao precisaria por private, em class e default
float x; //sao ocultos por default
float y; //sao ocultos por default
public: //daqui em diante tudo e acessivel.
void inicializa(float a,float b)
{ x=a; y=b; } //as funcoes de uma classe podem acessar os atributos private dela mesma.
void mostra(void)
{cout << "X:" << x << " , Y:" << y << endl;} };
```

```
void main()
{ ponto ap; //instanciacao
ap.inicializa(0.0,0.0); //metodos public
ap.mostra(); //metodos public }
```

Resultado do programa:
X:0 , Y:0
Comentários:

Este programa não deixa você tirar o ponto de (0,0) a não ser que seja chamada inicializa novamente. Fica claro que agora, encapsulando x e y precisamos de mais métodos para que a classe não tenha sua funcionalidade limitada.

Novamente: escrever ap.x=10; em main é um erro! Pois x está qualificada como private. Leia os exercícios.

Exercícios:

1) Implemente os métodos void altera_x(float a) , float retorna_x(void), void move (float dx,float dy); .Implemente outros métodos que achar importantes exemplo void distancia(ponto a) { return dist(X,Y,a.X,a.Y); }, onde dist representa o conjunto de operações matemáticas necessárias para obter a distância entre (X,Y) (a.X,a.Y). Você provavelmente usará a funcao sqr() da "library" <math.h> que define a raiz quadrada de um float.
Veja que no método distancia, podemos acessar os dados membro X e Y do argumento a, isto é permitido porque distancia é uma função membro da mesma classe de a, embora não do mesmo objeto, uma maneira de prover este tipo de acesso para outras classes é dotar a classe ponto de métodos do tipo float retorna_x(void);. Friends é um tópico avançado sobre encapsulamento que lida com visibilidade entre classes distintas.
Quando você precisar de outras funções matemáticas (seno, coseno), lembre-se da "library" <math.h>, é só pesquisar o manual do compilador ou usar ajuda sensível ao contexto para <math.h> ou até mesmo dar uma olhadinha na interface dela, no diretório das "libraries", alias essa é uma boa maneira de aprender sobre como implementa-las.

2) Retorne ao início deste tópico e releia as definições equivalentes de classes declaradas com struct e com class. Refaça o programa exemplo anterior usando struct, use encapsulamento em outros programas que você implementou.

Um dado membro é Public

Este programa é uma variante do anterior, a única diferença é que Y é colocado na parte public da definição da classe e é acessado diretamente. Além disso fornecemos aqui a função membro move, para que você possa tirar o ponto do lugar.

```
#include <iostream.h>
class ponto
{
float x; //sao ocultos por default
```

```
 public: //daqui em diante tudo e acessivel.
 ponto(float a,float b); //construtor tambem pode ser inline ou nao
 void mostra(void);
 void move(float dx,float dy);
 float y; //* Y nao e' mais ocultado};
 ponto::ponto(float a,float b)
{x=a;
 y=b;}
void ponto::mostra(void)
{cout << "X:" << x << " , Y:" << y << endl;}
 void ponto::move(float dx,float dy)
{x+=dx;
 y+=dy;}
void main()
{ponto ap(0.0,0.0);
 ap.mostra();
 ap.move(1.0,1.0);
 ap.mostra();
 ap.y=100.0;
 ap.mostra();}
```

Resultado do programa:
X:0 , Y:0
X:1 , Y:1
X:1 , Y:100
Comentários:

Observe que agora nada impede que você acesse diretamente y: ap.y=100.0, porém ap.x=10.00 é um erro. Observe em que parte (área) da classe cada um desses dados membro foi declarado.

Exercícios:

1)Crie os métodos float retorna_x(void), void altera_x(float a); que devem servir para retornar o valor armazenado em x e para alterar o valor armazenado em x respectivamente. Crie também as respectivas funções retorna e altera para todos os outros atributos.

Compilando um Programa com Vários Arquivos

Compilando um programa dividido em vários arquivos. Normalmente os programas C++ são divididos em arquivos para melhor organização e encapsulamento, porém nada impede que o programador faça seu programa em um só arquivo. O programa exemplo da classe ponto poderia ser dividido da seguinte forma:

```
//Arquivo 1 ponto.h, definicao para a classe ponto.
class ponto
{public: //daqui em diante tudo e acessivel.
void inicializa(float a,float b);
void mostra(void);
private:
float x; //sao ocultos por default
float y; //sao ocultos por default};
```

```
//Arquivo 2 , ponto.cpp , implementacao para a classe ponto.
#include <iostream.h>
#include "ponto.h"
void ponto::inicializa(float a,float b)
{ x=a; y=b; }
//as funcoes de uma classe podem acessar os atributos private dela mesma.
void ponto::mostra(void)
{cout << "X:" << x << " , Y:" << y << endl;}

//Arquivo 3 . Programa principal: princ.cpp
#include "ponto.h"
void main()
{ ponto ap; //instanciacao
ap.inicializa(0.0,0.0); //metodos public
ap.mostra(); //metodos public}
```

Os arquivos com extensão .h indicam header files. É costume deixar nesses arquivos somente a interface das classes e funções para que o usuário possa olhá-lo, como numa "library". Note que a parte public da classe foi colocada antes da parte private, isto porque normalmente é essa parte da definição da classe que interessa para o leitor.

No nosso caso o único arquivo `header' (.h) que temos é o "ponto.h" que define a classe ponto. Caso as dimensões de seu programa permitam, opte por separar cada classe em um `header file', ou cada grupo de entidades (funções, classes) relacionadas.

O arquivo 2 "ponto.cpp" é um arquivo de implementação. Ele tem o mesmo nome do arquivo "ponto.h" , embora isto não seja obrigatório. É mais organizado ir formando pares de arquivos "header / implementation".

Este arquivo fornece o código para as operações da classe ponto, daí a declaração: #include "ponto.h". As aspas indicam que se trata de um arquivo criado pelo usuário e não uma "library" da linguagem como <iostream.h>, portanto o diretório onde se deve encontrar o arquivo é diferente do diretório onde se encontra <iostream.h>.

O arquivo 3 "princ.cpp" é o arquivo principal do programa, não é costume relacionar seu nome com nomes de outros arquivos como no caso do arquivo 2 e o arquivo 1.

Observe que o arquivo 3 também declara #include "ponto.h", isto porque ele faz uso das dos tipos definidos em "ponto.h" , porém "princ.cpp" não precisa declarar #include <iostream.h> porque este não usa diretamente as definições de iostream em nenhum momento, caso "princ.cpp" o fizesse, o include <iostream> seria necessário.

O leitor encontrará em alguns dos exemplos seguintes as diretivas de compilação. Quando da elaboração de uma library para ser usada por outros programadores, não se esqueça de usá-las:

```
#ifndef MLISTH_H
#define MLISTH_H
//Codigo
#endif
#ifndef MLISTH_H
```

```
#define MLISTH_H
//defina aqui seu header file.
//perceba que "Nomearq.h" e escrito na diretiva como NOMEARQ_H
#endif
```

Essas diretivas servem para evitar que um header file seja incluído mais de uma vez no mesmo projeto. O seu uso se dá da seguinte forma:

Diagrama sugestão para organização de programas relativamente simples:

Saber compilar programas divididos em vários arquivos é muito importante. Isto requer um certo esforço por parte do programador, porque os métodos podem variar de plataforma para plataforma. Pergunte para um programador experiente de seu grupo.

Se o seu compilador é de linha de comando, provavelmente você poderá compilar programas divididos em vários arquivos usando makefiles. Já se seu compilador opera num ambiente gráfico, esta tarefa pode envolver a criação de um projeto.

Considerações C++:

Neste tópico iremos explicar recursos mais particulares de C++, porém não menos importantes para a programação orientada a objetos na linguagem. Alguns recursos como const que está relacionado com encapsulamento poderão ser retirados deste tópico em novas versões do tutorial.

Const

Este exemplo mostra o uso de funções const e sua importância para o encapsulamento. Const pode qualificar um parâmetro de função (assegurando que este não será modificado), uma função membro (assegurando que esta não modifica os dados membro de sua classe), ou uma instância de objeto/tipo (assegurando que este não será modificado.)

Os modos de qualificação descritos atuam em conjunto, para assegurar que um objeto const não será modificado, C++ só permite que sejam chamados para este objeto funções membro qualificadas como const. Const é também um qualificador, "specifier".

```
#include <iostream.h>
#include <math.h>
//double sqrt(double x); de math.h retorna raiz quadrada do numero
//double pow(double x, double y); de math.h calcula x a potencia de y
 const float ZERO=0.0;
 class ponto
{private:
 float x; //sao ocultos por default nao precisaria private mas e' bom
 float y; //sao ocultos por default
public: //daqui em diante tudo e acessivel em main.
ponto(float a,float b)
{ x=a; y=b; }
void mostra(void) const
{cout << "X:" << x << " , Y:" << y << endl;}
float distancia(const ponto hi) const
{return
```

```
 float(
 sqrt(
(pow(double(hi.x-x),2.0) + pow(double(hi.y-y),2.0))));
        //teorema de Pitagoras}};
void main()
{ ponto ap(3.0,4.0); //instanciacao
ap.mostra(); //funcoes membro public
const ponto origem(ZERO,ZERO); //defino objeto constante
origem.mostra();
cout << "Distancia da origem:" << origem.distancia(ap);}
```

Resultado do programa:

X:3 , Y:4
X:0 , Y:0
Distancia da origem:5

Comentários:

Const qualificando instâncias de objetos/tipos:

```
const ponto origem(ZERO,ZERO);
const float ZERO=0.0;
```

Const qualificando funções:

float distancia(const ponto hi) **const**;

Const qualificando argumentos:

float distancia(**const** ponto hi) const;

Exercícios:

1)Use tudo o que você aprendeu sobre const na classe reta, definida em CONSTRUTORES E AGREGAÇÃO.

Funções Inline

Requisitos:

Saber como um programa se comporta na memória, em termos de chamadas de função e gerenciamento da pilha, "stack".

O que são funções inline: Imagine uma chamada de uma função membro void altera_raio(float a) da classe circulo já apresentada, ac.altera_raio(a). Esta chamada envolve a passagem de parâmetros, inserção da função na pilha (stack), retorno de um valor (void), tudo isso representa uma diminuição da velocidade do programa com relação a um simples: ac.raio=a; que nesse caso funcionaria muito bem.

Ocorre que nós desejamos usar chamadas de métodos, vimos inclusive meios de esconder, encapsular o atributo raio para que a única alternativa para alterar seu valor seja através da chamada de altera_raio(a). Porque programar desta forma? Porque é mais seguro, mais próximo dos princípios de orientação a objetos. Em verdade POO se caracteriza por muitas chamadas de métodos e uso do "heap", área de memória usada pela alocação dinâmica.

O que as funções declaradas como inline fazem é traduzir a chamada do método em tempo de compilação em um equivalente ac.raio=17.0. evitando todo o contratempo descrito. Essa tradução do método é colocada na sequência de código do programa, você pode ter vários trechos que chamariam funções, desviariam o fluxo do programa até o retorno desta, convertidos em instruções

simples. Como desvantagem temos o aumento do tamanho do programa, visto que passarão a existir várias cópias diferentes da função no programa (uma para cada argumento) ao invés de um só protótipo de função que era colocado na pilha no momento da chamada e então tinha os argumentos substituídos.

Nos programa anteriores sobre a classe círculo, se a função membro mostra fosse inline haveria uma conversão da chamada interna (dentro de mostra) de retorna_raio()em simplesmente ac.raio. Pode haver conversão de várias funções inline aninhadas, estas conversões são seguras, porque são feitas pelo compilador.

Normalmente é vantajoso usar inline para funções pequenas que não aumentem muito o tamanho do programa ou funções onde velocidade é crucial. Aqui vale a conhecida regra 80:20, oitenta por cento do tempo do programa é gasto em vinte por cento dos métodos. Porém o programador não precisa se preocupar muito com funções inline na fase de desenvolvimento, este é um recurso C++ para aumento de eficiência que pode muito bem ser deixado para o final do projeto. Saiba porém que as diferenças de tempo decorrentes de seu uso são sensíveis.

Este exemplo explora as possibilidades que temos para declarar funções membro e como declará-las para que sejam do tipo inline:

```
#include <iostream.h>
struct ponto
{float x;
 float y;
void inicializa(float a,float b)
{ x=a; y=b; } //Apesar de nao especificado compilador tenta
//expandir chamada da funcao como inline porque esta dentro da definicao da classe.
void mostra(void); //com certeza nao e' inline, externa a classe e sem qualificador.
inline void move(float dx,float dy); //e' inline , prototipo, definicao };
void ponto::mostra(void)
{cout << "X:" << x << " , Y:" << y << endl;}
inline void ponto::move(float dx,float dy) //implementacao, codigo
{x+=dx;
 y+=dy;}
void main()
{ponto ap;
 ap.inicializa(0.0,0.0);
 ap.mostra();
 ap.move(1.0,1.0);
 ap.mostra();
 ap.x=100.0;
 ap.mostra();}
```

Comentários:

O compilador tenta converter a função inicializa em inline, embora não esteja especificado com a palavra reservada inline que isto é para ser feito. Esta é uma regra, sempre que a função estiver definida na própria classe (struct{}) o compilador tentará convertê-la em inline. Foi dito que o compilador tenta porque isto pode variar de compilador para compilador, se ele não consegue converter em inline ,

devido a complexidade da função, você é normalmente avisado, tendo que trocar o lugar da definição da função membro.
Note que se a função membro é implementada fora da classe, tanto o protótipo da função membro, quanto a implementação devem vir especificados com inline para que a conversão ocorra.

Resultado do programa:

X:0 , Y:0
X:1 , Y:1
X:100 , Y:1

Alocação Dinâmica com New e Delete

Neste tópico serão apresentados os recursos de C++ para alocação dinâmica de variáveis de tipos simples e objetos, ou seja, estudaremos a alocação de estruturas em tempo de execução.

Ponteiros, "Pointer"

Este exemplo mostra como trabalhar com ponteiros para variáveis de tipos pré-definidos (não definidos pelo usuário) usando new e delete.
O programa a seguir cria um ponteiro para uma variável inteira, aloca memória para esta variável e imprime seu valor.

```
#include <iostream.h>
void main()
{int* a; //declara um ponteiro para endereco de variavel inteira
a=new int(3); //aloca memoria para o apontado por a, gravando neste o valor 3
cout << (*a) << endl ; //imprime o valor do apontado por a
delete a; //desaloca memoria }
```

Resultado do programa:

3

Comentários:

Se a fosse uma struct ou class e tivesse um dado membro chamado dd, poderíamos obter o valor de dd através de (*a).dd que pode ser todo abreviado em a->dd onde -> é uma seta, ou flecha que você pode ler como "o apontado" novamente. Os parênteses em (*a).dd são necessários devido a precedência do operador . com relação ao *. Esta sintaxe abreviada será bastante usada quando alocarmos dinamicamente objetos.

Observação:

```
int* a;
a=new int(3);
```

pode ser abreviado por:

```
int* a=new int(3);
```

Vetores criados estaticamente

O exemplo a seguir aloca estaticamente, em tempo de compilação, um vetor de inteiros com três posições. Em seguida, gravamos as três posições e as mostramos na tela:

```
#include <iostream.h>
void main()
{int a[3]; //aloca o vetor de tamanho 3, estaticamente
 a[0]=1; //atribui a posicao indice 0 do vetor
 a[1]=2; //atribui 2 a posicao indice 1 do vetor
 a[2]=3; //atribui 3 a posicao indice 2 do vetor
 cout << a[0] << " " << a[1] << " " << a[2] << endl; //mostra o vetor}
```

Resultado do programa:

1 2 3

Resumo da sintaxe de vetores:

```
int a[3]; //cria um vetor de inteiros a com tres posicoes, indices uteis de 0 ate 2
float b[9]; //cria um vetor de float b com nove posicoes, indices uteis de 0 ate 8
b[8]=3.14156295; //grava 3.1415... na ultima posicao do vetor b
if (b[5]==2.17) { /*acao*/} ; //teste de igualdade
```

Diagrama do vetor:

Perceba que a faixa útil do vetor vai de 0 até (n-1) onde n é o valor dado como tamanho do vetor no momento de sua criação, no nosso caso 3.

Nada impede que você grave ou leia índices fora dessa área útil, isso é muito perigoso, porque fora dessa área, o que você tem são outras variáveis de memória e não o espaço reservado para seu vetor. É perfeitamente aceitável, embora desastroso escrever em nosso programa `a[4]=3;`. O compilador calcula o endereço de memória da posição 4 com base na posição inicial do vetor e o tamanho do tipo alocado. Após calculado o endereço da posição 4 o valor 3 é copiado, apagando o conteúdo anterior!

Comentários:

Note que não estamos usando ponteiros neste exemplo e é por isso que o vetor é alocado estaticamente, em tempo de compilação, é também por este motivo que o argumento que vai no lugar do 3 no código `int a[3];` deve ser uma expressão constante e não uma variável.

Copia de Objetos com Vetores Alocados Estaticamente

No primeiro exemplo do TAD fração vimos que o compilador fornece cópia bit a bit para objetos, alertamos também sobre o perigo de usar este tipo de cópia em conjunto com alocação dinâmica. O exemplo seguinte mostra um caso onde esta cópia oferecida pelo compilador é segura, o tópico seguinte mostra um caso onde esta cópia não é segura, leia ambos para ter uma visão geral do assunto:

```
#include <iostream.h>
class vetor_tres
{public:
 int vet[3]; //vetor alocado estaticamente numa classe.
 vetor_tres(int a,int b,int c)
{ vet[0]=a; vet[1]=b; vet[2]=c; } //construtor do vetor
```

```
void mostra(void)
{ cout << vet[0] << " " << vet[1] << " " << vet[2] << endl;}
//funcao membro para mostrar o conteudo do vetor };
void main()
{ vetor_tres v1(1,2,3); //criacao de um objeto vetor.
  vetor_tres v2(15,16,17); //criacao de um objeto vetor.
  v1.mostra(); //mostrando o conteudo de v1.
  v2.mostra(); //mostrando o conteudo de v2.
  v2=v1; //atribuindo objeto v1 ao objeto v2.
  v2.mostra();//mostrando v2 alterado.
  v1.vet[0]=44;
  v1.mostra(); //mostrando o conteudo de v1.
  v2.mostra(); //mostrando o conteudo de v2. }
```

Resultado do programa:
1 2 3
15 16 17
1 2 3
44 2 3
1 2 3
Comentários:

Perceba que no caso de alocação estática, quando o tamanho do vetor é conhecido em tempo de compilação a cópia é segura. Por cópia segura entenda: as posições do vetor são copiadas uma a uma e os objetos não ficam fazendo referência a um mesmo vetor, isto pode ser visto no resultado do programa, quando alteramos a cópia de v1, v1 não se altera.

Vetores criados dinamicamente

O exemplo a seguir os vetores são alocados dinamicamente, ou seja, você determina em tempo de execução qual o tamanho do vetor, Pascal não permite isso.

```
#include <iostream.h>
void main()
{ int tamanho; //armazena o tamanho do vetor a criar.
  int* vet; //ponteiro para inteiro ou vetor de inteiro ainda nao criado
  cout << "Entre com o tamanho do vetor a criar";
  cin >> tamanho;
  vet=new int[tamanho];
//alocando vetor de "tamanho" posicoes comecando em   a[0]
for (int i=0;i<tamanho;i++)
{cout << "Entre com o valor da posicao " << i << ":";
 cin >> vet[i];
 cout << endl; }//loop de leitura no vetor
 for (int j=0;j<tamanho;j++)
{cout << "Posicao " << j << ":" << vet[j]<<endl;}
//loop de impressao do vetor}
```

Resultado do programa:
Entre com o tamanho do vetor a criar3
Entre com o valor da posicao 0:1

Entre com o valor da posicao 1:2
Entre com o valor da posicao 2:3
Posicao 0:1
Posicao 1:2
Posicao 2:3

Comentários:

int* a;
Declara um ponteiro para inteiro.
a=new int[10];

Diferente de new int(10); os colchetes indicam que é para ser criado um vetor de tamanho 10 e não uma variável de valor 10. Ao contrário de alocação estática, o parâmetro que vai no lugar do valor 10 não precisa ser uma expressão constante.

int* a;
a=new int[10];

equivale a forma abreviada:

int* a=new int[10];

A faixa de índices úteis do vetor novamente vai de 0 até (10-1) ou (n-1).

Copia de Objetos com Vetores Alocados Dinamicamente

Essa determinação do tamanho do vetor em tempo de execução vai tornar a cópia de objetos feita pelo compilador diferente, ele vai copiar o ponteiro para o vetor, ou seja os objetos passam a compartilhar a estrutura na memória, o que nem sempre pode ser desejável! Existem maneiras de redefinir esta cópia feita pelo compilador o que veremos em 0.

```
#include <iostream.h>
class vetor_tres
{public:
 int* vet; //vetor alocado estaticamente numa classe.
 vetor_tres(int a,int b,int c)
{vet=new int[3];
 vet[0]=a;
 vet[1]=b;
 vet[2]=c; }//construtor do vetor
void mostra(void)
{ cout << vet[0] << " " << vet[1] << " " << vet[2] << endl;}
//funcao membro para mostrar o conteudo do vetor };
void main()
{ vetor_tres v1(1,2,3); //criacao de um objeto vetor.
  vetor_tres v2(15,16,17);//criacao de um objeto vetor.
  v1.mostra(); //mostrando o conteudo de v1.
  v2.mostra(); //mostrando o conteudo de v2.
  v2=v1; //atribuindo objeto v1 ao objeto v2.
  v2.mostra(); //mostrando v2 alterado.
  v1.vet[0]=44;
  v1.mostra(); //mostrando o conteudo de v1.
 v2.mostra(); //mostrando o conteudo de v2. }
```

Resultado do programa:

1 2 3

15 16 17
1 2 3
44 2 3
44 2 3

Comentários:

Note que quando alteramos a cópia de v1, v1 se altera. Isto ocorre porque o vetor não é copiado casa a casa, só se copia o ponteiro para a posição inicial do vetor, a partir do qual se calcula os endereços das posições seguintes v[3]==*(v+3).

Sobre o texto acima: "a partir do qual se calcula os endereços das posições seguintes", veja o programa exemplo:

```
#include <iostream.h>
void main()
{int* v;
 v=new int[3];
 cout << *(v+1)<<endl; //imprime o lixo contido na memoria de v[1]
 cout << v[1] <<endl; //imprime o lixo contido na memoria de v[1]
 //*(v)==v[0] é uma expressao sempre verdadeira }
```

Resultado do programa:
152
152

Comentários:

O que é importante deste exercício é que só se armazena a posição inicial do vetor, as outras posições são calculadas com base no tamanho do tipo alocado e regras de aritmética de ponteiros. Não podemos nos estender muito neste tema, leia sobre aritmética de ponteiros, porém para este tutorial, basta usar os vetores de forma que esta aritmética fique por conta da linguagem.

Vetores alocados dinamicamente e ponteiros são a mesma coisa, isto pode ser visto nesse programa exemplo que mistura a sintaxe de vetores e de ponteiros.

Referência &

Aqui vou ensinar passagem por referência, lembre-se que objetos são passados por referência, porque eles são referência, ponteiros.

Este tópico vai explicar como usar o operador &, também chamado de operador "endereço de...". Este operador fornece o endereço, a posição na memória de uma variável, de um argumento, etc. Sua utilização é muito simples, se a é uma variável inteira, &a retorna um ponteiro para a.

O programa a seguir ilustra a sintaxe do operador:

```
#include <iostream.h>
void main()
{ int a;
  a=10;
  int* p;
  p=& a;
(*p)=13;
cout << a; }
```

Explicando o programa passo a passo:
int a;
Declara uma variável inteira com nome a.
a=10;
atribui valor 10 a variável a.
int* p;
Declara um ponteiro de variável inteira , o nome do ponteiro é p.
p=& a;
Atribui o "endereço de a" , "& a" ao ponteiro p.
(*p)=13;
Atribui 13 ao "apontado de p", "(*p) ", mas como p aponta para a, a passa de valor 10 para valor treze através de p. P é um "alias" para a.
cout << a;
Imprime o valor esperado que é treze.
O programa a seguir usa o operador "endereço de" para modificar argumentos, parâmetros de uma função, ou seja utiliza passagem por referência, equivalente ao VAR de Pascal. Lembre-se que até agora os argumentos das nossas funções só eram passados por valor.

```
#include <iostream.h>
void incrementa(int& a)
{ a++; } //primeira funcao que usa passagem por referencia
void troca(int& a,int& b)
{ int aux=a;
  a=b;
  b=aux; }//segunda funcao que usa passagem por referencia

void main()
{int i1=10;
 int i2=20;
 incrementa(i1);
 cout << i1 << endl;
 troca(i1,i2);
 cout << i1 << endl; }
```

Resultado do programa:
11
20
Explicando passo a passo:
As funções criadas no programa, são como dissemos, capazes de alterar seus parâmetros, incrementa , incrementa seu único parâmetro e troca, troca os dois parâmetros inteiros.

Argumentos de Linha de Comando

Neste tópico vamos apresentar um exemplo que usa tudo o que foi aprendido até agora e introduz mais alguns outros conceitos. Este exemplo é um jogo de quebra cabeça. Provavelmente você já viu um quebra cabeça deste tipo, ele é chamado quebra cabeça de tijolos deslizantes. É composto de um tabuleiro

(matriz) quadrado preenchido com peças de 0 até (lado^2-1) onde lado é a dimensão de um lado do tabuleiro. A representação no programa é numérica, mas normalmente esses quebra cabeças são vendidos com desenhos pintados sobre as pecas. No lugar onde deveria estar a última peça deixa-se um espaço vazio para que se possa mover as peças do tabuleiro. O objetivo é colocar todos os tijolos em ordem, como no esquema abaixo:

Solucionado!

O que um quebra cabeça tem a ver com encapsulamento? Simples o jogador é obrigado a seguir as regras do jogo, portanto não pode conhecer todas as operações que se aplicam ao quebra cabeça, e também não pode acessar sua representação interna. Vamos ver como isso é feito usando as palavras reservadas private e public , conceitos de agregação e reuso de código de uma classe matriz. Além disso usamos const para garantir a integridade da matriz do quebra-cabeça.

Veja também que estamos abordando o tópico representação, não raro você terá que modelar objetos que representem algo do mundo real como motor, quebra-cabeça, conta bancária, circuito elétrico e assim por diante. Outros objetos podem ser mais abstratos, exemplo: árvore binária.

Mais sobre as regras:

Você concorda que dada uma situação onde o espaço vazio se encontra no meio do quebra-cabeça dado como exemplo o usuário só tem 4 opções? São elas: mover para cima (função membro movecima), mover para baixo , mover para a esquerda ou então para a direita. Note que essas funções membro só trocam peças vizinhas, isso é importante pois é provado matematicamente que algumas trocas de peças distantes ("Odd permutations") podem tornar o quebra-cabeça insolúvel. Isto é facil de ser verificado para um quebra cabeça 2x2.

Garantimos a solução do quebra-cabeça pelo caminho inverso do embaralhamento das peças, que parte do quebra-cabeça solucionado e aplica aleatoriamente sequências de movimentos iguais aos que o usuário usa para solucionar o jogo.

A matriz bidimensional é representada linearmente (vetor) e as funções membro de conversão de índice linear para colunas e/ou linhas e atribuições são fornecidas, mas somente as necessárias para o jogo .

Recursos utilizados:

Argumentos de linha de comando, funções de conversão da stdlib, representação linear de uma matriz, const.

Argumentos de linha de comando:

Você pode fazer programas que aceitem argumentos de linha de comandos, resultando em algo equivalente a:

```
dir /w //dos
format A: //dos
ou ls -la //unix
```

Esses argumentos podem ficar disponíveis na chamada da função main de seu programa como os parâmetros "argument counter" e "argument values", veja trecho de programa abaixo.

void main(int argc, char *argv[]) {.../*use argv e argc*/ ...}
argc é um inteiro
argv é um vetor de char* contendo os argumentos passados. (/w ...)
argv [0] é o nome do programa chamado.
A faixa útil de argvalues é : argv[1]...argv[argc-1]
Portanto se argc==2, temos um único argumento de linha de comando, disponível como string em argv[1] .

Herança

Hierarquias de tipos

Neste tópico mostraremos como construir hierarquias de tipo por generalização / especialização.

Uma hierarquia simples

Construiremos uma hierarquia de tipos simples para demonstrar herança pública em C++.

Comentários:

O diagrama acima representa a hierarquia de classes implementada e foi obtido a partir da janela de edição de uma ferramenta case para programação orientada a objetos.
A classe ponto que está no topo da hierarquia é chamada de classe base, enquanto que as classes ponto_reflete e ponto_move são chamadas classes filhas ou herdeiras. As classes da hierarquia são simples, ponto_move apresenta a função membro move, já vista neste tutorial, ponto_reflete apresenta a função membro (reflete) que inverte o sinal das coordenadas.
Dada a simplicidade das classes o leitor poderia se perguntar, porque não juntar as três em uma só . A pergunta faz sentido, mas e se quiséssemos criar uma classe ponto que não se movesse, apenas refletisse e outra que só se movesse? E se quiséssemos projetar nosso programa segundo uma hierarquia de especialização / generalização da classe ponto? O exemplo mostra como fazê-lo.

Herança Pública:

Na herança pública as classes filhas passam a ter as mesmas funções membro public da classe pai, as classes filhas podem acrescentar funções membro, dados membro e até redefinir funções membro herdadas (veremos mais tarde). Os atributos da classe pai não são acessíveis diretamente na classe filha a não ser que sejam qualificados como protected. Por isso é que se diz que as classes filhas

garantem pelo menos o comportamento "behaviour" da classe pai, podendo acrescentar mais características.
Diagrama de acesso, visibilidade, de dados membro e funções membro de uma classe pai para uma classe filha ou herdeira por herança pública:

Construtores e herança:

No construtor de uma classe filha o programador pode incluir a chamada do construtor da classe pai.

Destrutores e herança:

Quando um objeto da classe derivada é destruído, o destrutor da classe pai também é chamado dando a oportunidade de liberar a memória ocupada pelos atributos private da classe pai.

```
//header file
class ponto
{ private:
  float x; //sao ocultos por default
  float y; //sao ocultos por default
  public: //daqui em diante tudo e acessivel.
  ponto(float a,float b);
  void inicializa(float a,float b);
  float retorna_x(void);
  float retorna_y(void);
  void altera_x(float a);
  void altera_y(float b);
  void mostra(void); };
class ponto_reflete:public ponto //classe filha
{private: //se voce quer adicionar atributos...
 public:
 ponto_reflete(float a, float b);
 void reflete(void); };
class ponto_move:public ponto
{public:
 ponto_move(float a,float b);
 void move(float dx,float dy);};

//implementation file
#include <iostream.h>
#include "pontos.h"
ponto::ponto(float a,float b)
{ inicializa(a,b); }
void ponto::inicializa(float a,float b)

{x=a;
 y=b;}
float ponto::retorna_x(void)
{ return x; }
float ponto::retorna_y(void)
{ return y; }
void ponto::altera_x(float a)
```

```
{ x=a; }
void ponto::altera_y(float b)
{ y=b; }
void ponto::mostra(void)
{cout << "(" << x << "," << y << ")" <<endl; }
 ponto_reflete::ponto_reflete(float a,float b):ponto(a,b)
{ }
void ponto_reflete::reflete(void)
{altera_x(-retorna_x());
 altera_y(-retorna_y());}
 ponto_move::ponto_move(float a,float b):ponto(a,b)
{ }
void ponto_move::move(float dx,float dy)
{altera_x(retorna_x()+dx);
 altera_y(retorna_y()+dy);}

#include <iostream.h>
#include "pontos.h"
void main()
{ponto_reflete p1(3.14,2.17);
 p1.reflete();
 cout << "P1";
 p1.mostra();
 ponto_move p2(1.0,1.0);
 p2.move(.5,.5);
 cout << "P2";
 p2.mostra();}
```

Resultado do programa:
P1(-3.14,-2.17)
P2(1.5,1.5)

Herança Private:

Ainda não foi inserido no tutorial nenhum exemplo com este tipo de herança, mas o leitor pode experimentar mais tarde especificar uma classe herdeira por heranca private como: class herdeira: private nome_classe_base; , e notará que as funções membro desta classe base só são acessíveis dentro das declarações da classe filha, ou seja a classe filha não atende por essas funções membro, mas pode usá-las em seu código.
Herança private é um recurso que você precisa tomar cuidado quando usar. Normalmente, quando usamos herança dizemos que a classe filha garante no mínimo o comportamento da classe pai (em termos de funções membro) , a herança private pode invalidar esta premissa.
Muitos programadores usam herança private quando ficaria mais elegante, acadêmico, trabalhar com agregação. Uma classe pilha pode ser construída a partir de uma classe que implementa uma lista ligada por agregação ou por herança private. Na agregação (a escolhida em *hierarquias de implementação*) a classe pilha possui um dado membro que é uma lista ligada.

Protected

Igual ao exemplo um, mas agora tornando os atributos da classe pai acessíveis para as classes filhas através do uso de protected. Protected deixa os atributos da classe pai visíveis, acessíveis "hierarquia abaixo".
Diagramas de acesso, visibilidade, de dados membro e funções membro de uma classe pai para uma classe filha ou herdeira:
Para uma classe filha por herança

pública:
Visibilidade da classe herdeira
para o restante do programa:

```
//header file
class ponto
{protected: //*****aqui esta a diferenca ******
 float x; //visiveis hierarquia abaixo
 float y; //visiveis hierarquia abaixo
 public: //daqui em diante tudo e acessivel.
 ponto(float a,float b);
 void inicializa(float a,float b);
 float retorna_x(void);
 float retorna_y(void);
 void altera_x(float a);
 void altera_y(float b);
 void mostra(void);};
class ponto_reflete:public ponto
{private: //se voce quer adicionar dados membro encapsulados...
 public:
 ponto_reflete(float a, float b);
 void reflete(void);};
 class ponto_move:public ponto
{public:
 ponto_move(float a,float b);
 void move(float dx,float dy);};

//implementation file
#include <iostream.h>
#include "pontos.h"
ponto::ponto(float a,float b)
{inicializa(a,b);}
void ponto::inicializa(float a,float b)
{x=a;
 y=b;}
 float ponto::retorna_x(void)
{ return x; }
float ponto::retorna_y(void)
{ return y; }
void ponto::altera_x(float a)
{ x=a; }
void ponto::altera_y(float b)
{ y=b; }
void ponto::mostra(void)
```

```
{cout << "(" << x << "," << y << ")" <<endl; }
 ponto_reflete::ponto_reflete(float a,float b):ponto(a,b)
{ }
void ponto_reflete::reflete(void)
{x=-x; //*** protected da esse tipo de acesso aos atributos da classe pai
 y=-y; }
ponto_move::ponto_move(float a,float b):ponto(a,b)
{ }
void ponto_move::move(float dx,float dy)
{
x=x+dx; //acesso so na hierarquia, no resto do programa nao.
y=y+dy;}

#include <iostream.h>
#include "pontos.h"
void main()
{ponto_reflete p1(3.14,2.17);
 p1.reflete();
 cout << "P1";
 p1.mostra();
 ponto_move p2(1.0,1.0);
 p2.move(.5,.5);
 cout << "P2";
 p2.mostra();}
```

Redefinição de Funções Membro Herdadas

Igual ao exemplo anterior, mas agora redefinindo a função membro mostra para a classe filha ponto reflete.

Comentários:

Na verdade este exemplo deveria pertencer ao tópico de polimorfismo, contudo, nos exemplos seguintes usaremos também redefinições de funções membro, portanto faz-se necessário introduzi-lo agora. Teremos mais explicações sobre o assunto.

Uma classe filha pode fornecer uma outra implementação para uma função membro herdada, caracterizando uma redefinição "overriding" de função membro. Importante: a função membro deve ter a mesma assinatura (nome, argumentos e valor de retorno), senão não se trata de uma redefinição e sim sobrecarga "overloading".

No nosso exemplo a classe ponto_reflete redefine a função membro mostra da classe pai, enquanto que a classe herdeira ponto_move aceita a definição da função membro mostra dada pela classe ponto que é sua classe pai.

```
//header file
class ponto
{private:
 float x; //sao ocultos por default
 float y; //sao ocultos por default
 public: //daqui em diante tudo e acessivel.
 ponto(float a,float b);
 void inicializa(float a,float b);
 float retorna_x(void);
```

```
 float retorna_y(void);
 void altera_x(float a);
 void altera_y(float b);
 void mostra(void);};
class ponto_reflete:public ponto
{private: //se voce quer adicionar dados membro
 public:
 ponto_reflete(float a, float b);
 void reflete(void);
 void mostra(void);//redefinicao};
class ponto_move:public ponto
{public:
 ponto_move(float a,float b);
 void move(float dx,float dy);//esta classe filha nao redefine mostra };

//implementation file
#include <iostream.h>
#include "pontos.h"
ponto::ponto(float a,float b)
{inicializa(a,b);}
 void ponto::inicializa(float a,float b)
{x=a;
 y=b;}
float ponto::retorna_x(void)
{ return x; }
float ponto::retorna_y(void)
{ return y; }
void ponto::altera_x(float a)
{ x=a; }
void ponto::altera_y(float b)
{ y=b; }
void ponto::mostra(void)
{cout << "(" << x << "," << y << ")" <<endl; }
ponto_reflete::ponto_reflete(float a,float b):ponto(a,b)
{ }
void ponto_reflete::reflete(void)
{altera_x(-retorna_x());
 altera_y(-retorna_y());}
void ponto_reflete::mostra(void)
{cout << "X:" << retorna_x() << " Y:";
 cout << retorna_y() << endl;}//somente altera o formato de impressao
 ponto_move::ponto_move(float a,float b):ponto(a,b)
{ }
void ponto_move::move(float dx,float dy)
{altera_x(retorna_x()+dx);
 altera_y(retorna_y()+dy);}

#include <iostream.h>
#include "pontos.h"
void main()
{ponto_reflete p1(3.14,2.17);
 p1.reflete();
 cout << "P1";
 p1.mostra();
 ponto_move p2(1.0,1.0);
 p2.move(.5,.5);
```

```
 cout << "P2";
 p2.mostra();}
```

Resultado do programa:
P1X:-3.14 Y:-2.17
P2(1.5,1.5)

Outro exemplo:

```
const TAM = 80;
class BasAg
{
     protected :
                    char nome[TAM];
                    char numag[4];
     public :
                   BasAg() { nome[0] = '\0'; numag[0] = '\0'; }
                   BasAg(char n[],char ng[]) {}
                   void print(){}
};

class Agente : public BasAg // herança ou classe derivada
{
   protected :
                   float altura;
                   int idade;
   public :
                   Agente() : BasAg() { altura = 0; idade = 0; }
                   Agente(char n[], char ng[], float a, int i) :
                          BasAg(n,ng) { altura = a; idade = i; }
                   void print();

};

void main()
{
  Agente agente("Daniel","1",1.78,21);
  agente.print();
}
```

No exemplo acima, o método "print()" da classe derivada agente é reescrita com o mesmo nome do método da classe base e, ainda, chama o método "print()" da classe base BasAg através do operador de escopo "::". Quando for executado o comando "agente.print()" do exemplo acima, o compilador executará o método print() da classe agente. Mas, o primeiro comando deste método é chamar o método de mesmo nome da classe base, o compilador executa este método e, após os comandos do método da classe derivada.

Exercício:

1)Teste redefinição de função membro colocando "cout's" em funções membro da hierarquia, tais como: `cout << "Redefinido!"; cout << "Original!";`

Herança Pública e Privada

Quando criamos uma classe derivada com a clausula "public", estamos dizendo que os dados públicos da classe base serão dados públicos na classe derivada e, os dados protegidos serão os dados protegidos da classe derivada. A classe derivada não terá acesso aos dados privados. Podemos criar uma classe com a clausula "private", com isso tanto os dados públicos como protegidos da classe base serão dados privados da classe derivada. Com isso, a classe derivada não terá acesso a nenhum dado da classe base.

```
class BASE
{
   protected : Int secreto;
   private : int ultra_secreto;
   public : int publico;
};
class DERIV1 : public BASE
{
   public :
            int a = secreto; // ok
            int b = ultra_secreto; // erro :não-acessível
            int c = publico // ok
};
class DERIV2 : private BASE
{
   public :
            int a = secreto; // ok
            int b = ultra_secreto; // erro :não-acessível
            int c = publico // ok
} ;
```

Herança Múltipla

Quando uma classe herda as características de mais de uma classe-base. Abaixo, temos o exemplo de herança múltipla. Declaramos as classes simples Cadastro, Imóvel e Tipo. Declaramos a classe Venda como herança múltipla das três anteriores.

```
class Cadastro
{
  private :
              char nome[30], fone[20];
  public :
              Cadastro();
              void getdata();
              void putdata();
};
class Imóvel
{
  private :
```

```
        char end[30], bairro[20];
        float AreaUtil, AreaTotal;
        int quartos;
  public :
        Imóvel();
        void getdata();
        void putdata();
};
class Tipo
{
  private :
        char tipo[20]; // Residencial, Loja, Galpão...
  public :
        Tipo();
        void getdata();
        void putdata();
};
class Venda : private Cadastro,Imovel,Tipo
{
 private :
        float valor;
 public :
        Venda() : Cadastro(), Imóvel(), Tipo();
        void getdata();
        void putdata();
};
```

Funções Amigas

Por default, os membros privados de uma classe somente podem ser acessados por outros membros da mesma classe. Contudo, uma classe pode dar permissão para uma função não membro desta classe acessar os seus membros privados, mas não estará associada a um objeto da classe. Isto é feito com a declaração friend.
Uma função amiga pode agir em duas ou mais classes diferentes, fazendo o papel de elo de ligação entre as classes.

Exemplo :

```
#include <iostream.h>
class tempo
{
   private :
                long segundos;
   public : tempo(long h, long m, long s) { segundos = h * 3600 + m * 60
+s; }
               friend char *prntm(tempo); // declaração amiga
};
char *prntm(tempo tm) <
{
   char * buff = new char[9]; // new executa uma alocação
                dinâmica  ao invés de ser alocado na    pilha
   int h = tm.segundos / 3600;
   int resto = tm.segundos % 3600;
   int m = resto / 60;
   int s = (resto % 60);
   buff[0] = h / 10 + '0';
```

```
    buff[1] = h % 10 + '0';
    buff[2] = ':';
    buff[3] = m / 10 + '0';
    buff[4] = m % 10 + '0';
    buff[5] = ':';
    buff[6] = s / 10 + '0';
    buff[7] = s % 10 + '0';
    buff[8] = '\0';
    return buff;
}
void main()
{
    tempo tm(5,8,20);
    cout << "\n" << prntm(tm);
}
```

No exemplo acima, a função prntm() tem acesso ao dado privado da classe tempo. Esta declaração pode ser declarada como pública ou privada, pois, não faz diferença o local onde é declarada.
Além de ser possível declarar funções independentes como amigas, podemos declarar uma classe toda como amiga de outra. Os métodos da classe serão todas amigas da outra classe. A diferença é que estas funções-membro tem também acesso a parte privada ou protegida de sua própria classe.
Exemplo:

```
#include <iostream.h>
class tempo
{
   private : long h,m,s;
   public : tempo (int hh,int mm,int ss) { h = hh; m = mm; s = ss; }
            friend class data; //data é uma classe amiga
};
class data
{
  private : int d,m,a;
  public :
            data (int dd,int mm,int aa) { d = dd; m = mm; a = aa;}
            void prndt (tempo tm)
             {
                cout << "\nData: " <<d<<'/'<<m<<'/' <<a;
                cout << "\nHora: " <<tm.h<< ':'<<tm.m<< ':' <<tm.s;
             }
};

void main()
{

   tempo tm(12,18,30);
   tempo tm1 (13,22,10);
   data dt(18,12,55);
   dt.prndt(tm);
   dt.prndt(tm1);
}
```

No exemplo acima, na classe tempo, declaramos toda a classe data como amiga. Então todas as funções-membro de data podem acessar os dados privados de tempo. A palavra chave friend

oferece acesso em uma só direção, pois, como no exemplo acima, a classe tempo é amiga da classe data, mas o contrário não é verdadeiro.

Sobrecarga de Operadores

Sobrecarga de operadores é uma das opções mais poderosas da orientação a objeto. Sobrecarregar um operador significa alterar o domínio de um operador pré-definido de uma linguagem de operação. Por exemplo, o domínio do operador `+' é os números reais. Se desejarmos que o operador `+' ao invés de somar dois números inteiros e passe a somar duas matrizes de dimensões nxn, devemos declarar uma função-membro `operator +' dentro da classe matriz.

À partir de agora, cada vez que o operador `+' for executado entre dois objetos da classe matriz ele irá somá-las e atribuir o resultado ´a uma terceira matriz. Sobrecarregar um operador significa redefinir o seu símbolo, de modo que ele se adeque aos tipos abstratos de dados definidos pelo usuário, tais como, estruturas e classes. A implementação de sobrecargas de operadores é definido por meio de funções chamadas operadoras. Estas funções podem ser criadas como membros de classe ou como funções globais.
A sobrecarga de operadores deve respeitar a definição original do operador. Por exemplo, o operador soma necessita de dois argumentos, não podemos modificá-lo para três argumentos. Para realizar a sobrecarga devemos usar os operadores definidos, ou seja, não podemos criar operadores novos. A sobrecarga deve obedecer à precedência original do operador sobrecarregado. Não é possível modificar a precedência dos operadores sobrecarregados.

Os operadores ".", "::" e "?:" não podem ser sobrecarregados. Operadores unários ("++", "--" e "-"), definidos como métodos de classes, não recebem nenhum argumento, enquanto que os operadores binários recebem um único argumento. Podemos sobrecarregar além dos operadores unários e binários, strings.

Podemos usar os operadores "=", "+"e "+=" para concatenar e comparar strings.

Sobrecarga de Operadores Unários

Sobrecarga do operador de incremento pré-fixado tem a seguinte sintaxe : void operator ++() { comandos } void operator --() { comandos } A função não recebe nenhum argumento e não retorna nada neste caso. Esta declaração indica ao compilador que este método deve ser executado cada vez que for usado este operador com objetos desta classe. A chamada pode ser duas maneiras : ++p1; ou p1.operator++(); A sobrecarga do operador de incremento pós-fixado tem a seguinte sintaxe : void operator ++(int) { comandos } void operator --(int) { comandos } O parâmetro "int" indica para o compilador que a função sobrecarregada é pós-fixada. Exemplo de um operador sobrecarregado pré-fixado.

```
#include <iostream.h>
class ponto
{
  private : int x,y;
  public :
          ponto(int x1=0,int y1=0) // construtor { x = x1; y = y1;}
          void operator ++() // função operadora prefixada { ++x; ++y; }
          void printpt() const // imprime ponto { cout << "("
                << x << "," << y << ")"; }
};
```

```
void main()
{
    ponto p1, p2(2,3), p3; // declara e inicializa
    cout << "\n p1 = ";
    p1.printpt();
    cout << "\n p2 = ";
    p2.printpt(); ++p1; //incrementa p1
    ++p2; // incrementa p2
    cout << "\n ++p1 = ";
    p1.printpt();
    cout << "\n ++p2 = ";
    p2.printpt();
    p3 = p1;
    cout << "\n p3 = ";
     p3.printpt();
}
```

Resultado :
p1 = (0,0)
p2 = (2,3)
++p1 = (1,1)
++p2 = (3,4)
p3 = (1,1)

Sobrecarga de Operadores Binários

Quando operadores binários são sobrecarregados, a função operadora terá um argumento. A função operadora, quando declarada como método de uma classe sempre precisa de um argumento a menos que o número de operandos daquele operador.

Exemplo:

```
#include <iostream.h>


#include <iomanip.h>
class venda
{
   private :
            int npecas;
            float preco;
   public :
            venda() {}
            venda(int np, float p) // construtor com argumentos { npecas= np; preco = p; }
            void getvenda()
            {
              cout << "Insira Numero de Pecas: ";
              cin >> npecas;
              cout << "Insira Preco : ";
              cin >> preco;
            }
            venda operator + (venda v) const; // funçãooperador
            void printvenda() const;
};
venda venda::operator + (venda v) const // soma duas vendas
{
   int pec = npecas + v.npecas;
```

```
  float pre = preco + v.preco;
  return venda(pec,pre);
}
void venda::printvenda() const
{
  cout << setiosflags(ios::fixed) // nao notaçãocientifica
          << setiosflags(ios::showpoint) // ponto   decimal
          <<setprecision(2) // duas casas decimais
          << setw(10) << npecas;
                // tamanho do campo 10
  cout << setw(10) << preco << "\n";
}
void main()
{
  venda A(58,12734.53),B,C(30,6000.3),T,Total;
  B.getvenda();
  T = A + B;
  Total = A + B + C; //somas múltiplas
  cout << "Venda A ........";
  A.printvenda();
  cout << "Venda B ........";
  B.printvenda();
  cout << "Venda:A + B ....";
  T.printvenda();
  cout << "Totais:A + B + C";
  Total.printvenda();
  return;
}
```

Resultado:

```
Insira No.Peças: 45
Insira Preço: 10987.85
Venda: A.... 58 12734.53
Venda: B.... 45 10987.85
Venda A + B: ..... 103 23722.38
Totais A + B + C: 133 29722.68
```

Funções Virtuais

São utilizadas quando queremos fazer referência através de uma matriz de ponteiros para uma classe base que contém várias classes derivadas com o objetivo de acessar um mesmo método de cada classe derivada. Para utilizarmos este recurso da POO, declaramos os métodos que quisermos como funções virtuais da classe base. A sintaxe é a seguinte:

virtual <tipo> <nome método> <parâmetros>

Exemplo:

```
class base {
  public :
    virtual void print() {
           cout << "\nBASE";
    }
};
```

```
class deriv0 : public base {
  public :
    void print() {
          cout <<"\nDERIV0";
    }
};

class deriv1 : public base {
  public :
    void print() {
          cout <<"\nDERIV1";
    }
};
class deriv2 : public base {
  public :
    void print() {
          cout <<"\nDERIV2";
    }
};
void main() {
  base *p[3]; // matriz de ponteiro para a classe base
  deriv0 dv0; // objeto da classe deriv0 deriv1 dv1; // objeto da classe deriv1 deriv2 dv2;
              // objeto da classe deriv2
  p[0] = &dv0; p[1] = &dv1; p[2] = &dv2;
  for (int i = 0; i < 3; i++)
    p[i]-> print();
}
```

O resultado produzido é : DERIV0 DERIV1 DERIV2 Se não colocarmos a palavra virtual na declaração do método print() da classe base o resultado será: BASE BASE BASE

Funções Virtuais Puras

Criamos funções virtuais puras para o uso de herança de uma classe base que nunca terá um objeto definido. Esta função não possui código. Na declaração do protótipo da função usamos o operador de atribuição seguido de um zero após o seu protótipo. A função serve somente para prover uma interface polimórfica para as classes derivadas. Exemplo :

```
class base {
  public : virtual void print() = 0;
};

class deriv0 : public base {
  public :
    void print() {
          cout <<"\nDERIV0";
    }
};

class deriv1 : public base {
  public :
    void print() {
          cout <<"\nDERIV1";
    }
};
```

```
class deriv2 : public base {
   public :
      void print() {
              cout <<"\nDERIV2";
      }
};

void main() { base *p[3]; // matriz de ponteiro para a classe base
deriv0 dv0; // objeto da classe deriv0 deriv1 dv1; // objeto da classe
deriv1 deriv2 dv2; // objeto da classe deriv2
p[0] = &dv0; p[1] = &dv1; p[2] = &dv2;
for (int i = 0; i < 3; i++) p[i]-> print();
}
```

O resultado produzido é : DERIV0 DERIV1 DERIV2

A classe base definida acima, não pode ter objetos declarados é chamada de classe abstrata. Contudo, podemos definir um ponteiro para uma classe abstrata para manipular objetos de classes derivadas.

Gabaritos de funções

Considere a seguinte função que troca os valores de dois inteiros:

```
void swap (int& x, int& y) {
int tmp = x;
x = y;
y = tmp;
}
```

Considere agora a extensão desta rotina para fazer a troca de valores para *floats*, *longs*, ou objetos de quaisquer outros tipos -- a solução de repetir o código acima para todas é possível, mas certamente não deve ser a melhor.
Um gabarito de função permite que o computador faça esta repetição de código, e não o programador. O gabarito para a função *swap* seria definido como

```
template < class T >
void swap (T& x, T& y) {
T tmp = x;
x = y;
y =tmp;
}
```

Toda vez que a função *swap* for chamada com um dado par de tipos, o compilador C **++** irá até a definição acima e criará uma outra *função gabarito* como uma instância do gabarito de função acima. Funções gabaritos podem ser sobrepostas por outras definições para tipos específicos, se necessário.

Gabarito de classes

Um gabarito de classe permite definir um padrão para definições de classes. Assim como para gabaritos de funções, a declaração de um gabarito de classe é precedida por *templateclass T$" src="img11.gif" width=19 align=middle border=0*, onde *T* é apenas uma referência a tipo ou classe que será utilizada na declaração.
O seguinte exemplo ilustra a declaração de um gabarito de classe para declarar vetores de elementos para diversos tipos.

```
Template < class T >
class Vector {
T data;
int size;
public:
Vector(int);
Vector() { delete [] data; }
T& operator[] (int i) { return data[i]; }
};

// observe a sintaxe para definicao fora da classe
template < class T >
Vector < T >::Vector(int n) {
data = new T[n];
size = n;
};

// exemplo de declaracao
main () {
Vector < int > ix(5); // gera um vetor de inteiros
Vector < float > fx(6); // gera vetor de floats

//...
}
```

Observe que, ao contrário de funções gabaritos, classes gabaritos (as instâncias de gabaritos de classes) devem ser explícitas sobre os parâmetros sobre os quais elas irão instanciar.

Referências Bibliográficas

Cesta, André A., C++ Como uma linguagem de Programação Orientada a Objetos, UNICAMP, 1996.

Bueno, André D., Apostila de Programação Orientada a Objetos em C++, UFSC, 2002.

Stroustrup. B., “The C++ Programming Language”. Addison-Wesley 1991.