Classes e Objectos

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

ex:. Um programa que utiliza uma interface controladora de um motor eléctrico definiria a classe motor. Os atributos desta classe seriam: temperatura, velocidade, tensão aplicada. Estes 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.

Especificar uma Classe

Vamos supor um programa que controla um motor eléctrico através de uma saída serie. A velocidade do motor é proporcional à tensão aplicada, e esta proporcional aos bits que vão para saída serie que passam por um conversor digital analógico (DAC).

Vamo - nos abstrair de 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, inteiro (int).

Usaremos o número 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 série

O motor precisa conhecer a saída serie, a sua ligação com o "motor do mundo real". Vamos supor uma representação em hexadecimal do atributo endereço de porta serie, um possível nome para o *atributo: enderecomotor*.

Alteração do valor da velocidade

Internamente o utilizador da classe motor pode desejar alterar a velocidade, cria-se então a função :

void altera_velocidade(int novav);

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.

Métodos ou Funções Membro

O C++ permite que se acrescente funções de manipulação da *struct* na 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, diz - se que o objecto está a receber uma mensagem (para executar uma acção). Um programa simples para testes sobre funções membro seria o seguinte:

Exemplo 1 :

```
#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 esquecas os parenteses, é uma funcao membro e não atributo!

  cout << umcontador.num << endl;
  umcontador.incrementa();
  cout << umcontador.num << endl;
}
```

Resultado do programa:
0
1
Comentários: O programa define um objecto 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
comeca inicializa a contagem a 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 ganha acesso directo 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 objecto 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 objecto 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.

Exemplo 2 :

```
#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 objecto circulo (criação)
        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 objecto. O objecto tem as coordenadas x e y somadas com os argumentos dessa função membro. Nota que esta função membro representa uma maneira mais segura, clara, elegante de alterar as coordenadas do objecto do que acessá-las directamente 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 objectos, 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 objectos só precisam ter uma referência para esta tabela.

Funções Membro que retornam um valor

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 programador. Sendo assim, sua chamada no programa aplica - se 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*;

Exemplo 3 :

```
#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 função membro nao dado!
  cout << umcontador.retorna_num() << endl;
  umcontador.incrementa();
  cout << umcontador.retorna_num() << endl;
}
```

Resultado do programa:
0
1

Funções Membro Declaradas Externas à Classe, Funções Membro

Este exemplo apresenta a implementação, definição, das funções fora da declaração da *struct*. Além disso introduz uma nova função chamada *inicializa* e funções *float retorna_raio(void)*; e *void altera_raio(float a)*. *inicializa* coloca o ponto nas coordenadas passadas como seus argumentos. Introduzimos esta função membro aqui para preparar a explicação sobre construtores.

Comentários: Numa declaração de uma classe coloca - se normalmente a declaração das funções membro depois da declaração dos atributos, porém podemos fazer intercalações ou adoptar qualquer ordem que nos convenha.

O programador não é obrigado a implementar as funções membro dentro da declaração da classe, basta defini-las e apresentar a implementação em separado segundo a sintaxe (compilável) descrita a seguir:

Exemplo 4 :

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

```
struct teste
{
  int x;

  void altera_x(int v);
  //somente definicao implementacao vem depois, fora da classe
};

void teste::altera_x(int v) { x=v;}  //esta já é a implementacao codigo

void main()
{
  teste a; //instaciação de um objecto

  a.altera_x(10);
  //chamada da função membro com valor 10 que sera impresso a seguir

  cout << a.x; //imprime o dado membro
}
```

Resultado do programa anterior:
10

Exemplo 5 :
Programa exemplo círculo, mais complexo:

```
#include <iostream.h

//para cout

struct circulo
{
  float raio;
  float x;
  float y;

  void inicializa(float ax,float by,float cr);
  void altera_raio(float a);
  float retorna_raio(void);
```

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

void circulo::inicializa(float ax,float by,float cr)
{
  x=ax;
  y=by;
  raio=cr;
}

void circulo::altera_raio(float a)
{
  raio=a;
}

float circulo::retorna_raio(void)
{
  return raio;
}

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

void circulo::mostra(void)
{
  cout << "Raio:"<< retorna_raio() <<endl;
  cout << "X:"<<x << endl;
  cout << "Y:" <<y<< endl;
}

void main()
{
  circulo ac;

  ac.inicializa(0.0,0.0,10.0);
```

```
  ac.mostra();
  ac.move(1.0,1.0);
  ac.mostra();
  ac.x=100.0;
  ac.altera_raio(12.0);
  ac.mostra();
}
```

Comentários: Observe que a função membro *mostra* chama a função membro *float retorna_raio(void)* que é da mesma classe. Fica implícito da definição de *mostra* que *retorna_raio()* se aplica ao mesmo objecto instanciado que recebeu a chamada de *mostra*, ou seja, não é necessário usar o . na chamada de *retorna_raio()*. Em programas maiores, chamadas aninhadas de funções membro são bastante comuns.

Programação orientada a objectos e interfaces gráficas com o utilizador: Existem *libraries* de classes que permitem o programador C++ desenvolver aplicações para ambientes como o *Microsoft Windowsreg.* de uma maneira bastante abstracta, este é um exemplo claro de reuso de código, afinal o programador não precisa saber de detalhes da interface para programar nela.

Resultado do programa:

```
Raio:10
X:0
Y:0
Raio:10
X:1
Y:1
Raio:12.0
X:100.0
Y:1
```

C++: As classes em C++ englobam os dados membros e as funções membros. Para executar uma ação sobre o objecto ou relativa a este basta chamar uma função membro para este: ac.mostra();

A função membro não precisa de muitos argumentos, porque é própria da classe e portanto ganha acesso aos dados membro do objecto para o qual ela foi associada:

```
float circulo::retorna_raio(void)
{
  return raio; //tenho acesso direto a raio.
}
```

Segurança: Em C++ o programador pode aceder directamente os dados do tipo definido pelo usuário: *ac.x=100.0*.

Veremos maneiras de proibir em C++ este tipo de acesso directo ao dado membro, deixando este ser modificado somente pelas funções membro. Isto nos garante maior segurança e liberdade pois podemos permitir ou não o acesso para cada dado membro de acordo com nossa vontade.

Eficiência: Alguém pode argumentar que programas que usam bastante chamadas de funções podem se tornar pouco eficientes e que poderia ser melhor aceder diretamente os dados de um tipo definido pelo usuário ao envés de passar por todo o trabalho de cópia de argumentos, inserção da função no pilha, etc.

Na verdade não se perde muito em eficiência, e além disso muitas vezes não se deseja permitir sempre o acesso directo aos dados de um tipo definido pelo programador por razões de segurança. Nesse sentido o C++ oferece um recurso que permite ganhos em segurança sem perder muito em eficiência.

Constructores

Construtores são funções membro especiais chamadas pelo sistema no momento da criação de um objecto. Elas não possuem valor de retorno, porque você não pode chamar um construtor para um objecto. Construtores representam uma oportunidade de inicializar de forma organizada os objectos; imagina se tu te esqueces de inicializar correctamente ou o fazes duas vezes, etc.

Um construtor tem sempre o mesmo nome da classe e não pode ser chamado pelo utilizador 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 estatica 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 seguinte é simples, semelhante aos anteriores, presta atenção na função membro com o mesmo nome que a *classe (struct)*, este é o construtor:

Exemplo 6 :

```
#include <iostream.h

struct ponto
{
  float x;
  float y;

  public:

  ponto(float a,float b);
  //esse e o contrutor, 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: Nota que com a definição do construtor, és obrigado a passar os argumentos deste no momento da criação do objeto.

Constructores e Agregação

O programa exemplo deste tópico cria uma *classe recta* com dois dados membro da *classe ponto* este exemplo é o resultado do exercício anterior, com um recurso a mais de C++. C++ permite que no construtor da *classe recta*, você chame os construtores dos atributos da *classe ponto*, se não o fizeres o compilador acusará um erro, pois os atributos ponto possuem constructores e eles precisam ser chamados para que a inicialização se complete de modo correcto para o conjunto.

Observa o código do constructor da *classe recta* usado no exemplo:

```
recta(float x1,float y1,float x2,float y2):p1(x1,y1),p2(x2,y2)
{
  //nada mais a fazer, os construtores de p1 e p2 ja foram chamados
}
```

p1(x1,y1) e *p2(x2,y2)* são as chamadas dos constructores da *classe ponto*, elas devem ficar fora do *corpo {}* do constructor, nesta lista separada por vírgulas deves inicializar todos os atributos. Os tipos básicos como *int*, *float*, etc podem ser inicializados nessa lista.

Por exemplo se a *classe recta* tivesse um atributo inteiro de nome *identificação*, a lista poderia ser da seguinte forma:

```
reta(float x1,float y1,float x2,float y2):p1(x1,y1),p2(x2,y2),identificacao(10)
{
  //nada mais a fazer, os construtores de p1 e p2 ja foram chamados
}
```

seria como se identificação tivesse um constructor que tem como argumento o seu valor. ou

```
recta(float x1,float y1,float x2,float y2):p1(x1,y1),p2(x2,y2)
{
  identificacao=10;
  //tambem pode, porque tipos básicos (int) em C++ não são objectos
  // portanto nao tem construtores
}
```

Vamos ao exemplo, que novamente é semelhante aos anteriores, para que o leitor preste atenção somente nas mudanças, que são os conceitos novos, sem ter que se esforçar muito para entender o programa:

Exemplo 7 :

```
#include <iostream.h

struct ponto
{
  float x;
  float y; //coordenadas

  ponto(float a,float b)
  {
    x=a;
    y=b;
```

```
  } //construtor

  void move(float dx,float dy)
  { x+=dx; y+=dy; } //funcao membro comum

  void inicializa(float a,float b)
  { x=a; y=b; }

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

struct recta
{
  ponto p1;
  ponto p2;

 recta(float x1,float y1,float x2,float y2):p1(x1,y1),p2(x2,y2)
 {
    //nada mais a fazer, os contrutores de p1 e p2 ja foram chamados
  }

  void mostra(void);
};

void recta::mostra(void)
{
  p1.mostra();
  p2.mostra();
}

void main()
{
  recta r1(1.0,1.0,10.0,10.0); //instanciação da recta r1

  r1.mostra();
}
```

Resultado do programa:

X:1 , Y:1
X:10 , Y:10

Destructores

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

Não se pode chamar o destructor, o que se faz é fornecer ao compilador o código a ser executado quando o objecto é destruído, apagado. Ao contrário dos constructores, os destructores não tem argumentos.

Os destructores são muito úteis para "limpar a casa" quando um objecto 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 práctica e segura de organizar o uso do "heap". A importância dos destructores em C++ é aumentada pela ausência de "garbage collection" ou coleta automática de lixo.

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

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

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

Exemplo 8 :

```
 //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 codigo
  }

  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 objecto 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 destructor. 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 destructores 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 apresentaremos mais 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".

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

Protected está relacionada com herança, por agora vamos focalizar a 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 pensares no seguinte: esses qualificadores aplicam - se 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 não especificares 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 supôr que instanciaste (criaste) um objecto do tipo ponto em seu programa:

ponto meu; //instanciação

Segundo o uso de qualquer uma das definições da classe ponto dadas acima não podes escrever no teu programa:

meu.x=5.0; //erro !

,como fazias antes, a não ser que x fosse declarado depois de *public* na definição da classe o que não ocorre aqui. Mas podes 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 o que é de sua classe, veja o programa seguinte.

Você pode escrever: *meu.move(5.0,5.0);* ,porque a 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).

Atributos Private, Funções Membro Public

Aplicar encapsulamento a classe ponto definida anteriormente.

Exemplo 9 :

```
#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; //instanciação

  ap.inicializa(0.0,0.0); //métodos public
  ap.mostra(); //metodos public
}
```

Resultado do programa:
X:0 , Y:0

Comentários: Este programa não deixa 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 a sua funcionalidade limitada.

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

Um Dado Membro 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 directamente.

Além disso fornecemos aqui a função membro *move*, para que você possa tirar o ponto do lugar.

Exemplo 10 :

```
 #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: Observa que agora nada impede que acesses directamente y: ap.y=100.0, porém ap.x=10.00 é um erro. Observa em que parte (área) da classe cada um desses dados membro foi declarado.

Compilar um Programa com Vários Arquivos

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

Exemplo 11 :

```
//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
```

Tipo Abstracto de Dados

Tipo abstracto de dados, TAD, preocupa - se em proporcionar uma abstracção sobre uma estrutura de dados em termos de uma interface bem definida. São importantes os aspectos de encapsulamento, que mantém a integridade do objeto evitando acessos inesperados, e o facto de o código estar armazenado num só lugar o que cria um programa modificável, legível e coeso.

Uma classe implementa um tipo abstrato de dados.
São exemplos de tipos abstratos de dados:

1) Uma árvore binária com as operações usuais de inserção, remoção, busca ...

2)Uma representação para números racionais (numerador, denominador) que possua as operações aritméticas básicas e outras de conversão de tipos.

3)Uma representação para ângulos na forma (Graus, Minutos, Segundos). Também com as operações relacionadas, bem como as operações para converter para radianos, entre outras.

Tipo abstracto de dados é um conceito muito importante em programação orientada a objectos e por este motivo é logo apresentado neste tutorial. Os exemplos seguintes são simples devido ao facto de não podermos usar todos os recursos C++ por razões didácticas. Devido a esta importância, à medida em que formos introduzindo novos recursos exemplificaremos também com aplicações na implementação tipos abstratos de dados.

TAD Fracção

Tipo abstraxto de dados fração. Baseado no conceito de número racional do campo da matemática. Algumas operações não foram implementadas por serem semelhantes às existentes. Por equanto não podemos fazer uso de recursos avançados como sobrecarga de operador e templates.

Resumo das operações matemáticas envolvidas:

Simplificação de fracção: (a/b)=((a/mdc(a,b)) / (b/mdc(a,b)))
Onde mdc(a,b) retorna o máximo divisor comum de ab.

Soma de fração: (a/b)+(c/d)=((a.d+c.b) / b.d) simplificada.
Multiplicação de fração: (a/b) * (c/d)= ((a*c) / (b*d)) simplificada.

Igualdade: (a/b)== (c/d) se a*d == b*c.

Não igualdade: (a/b) != (c/d) se a*d b*c

Maior ou igual que: (a/b)(c/d) se a*d b*c

Tópicos abordados: Constructores em geral, destructores, tipo long, criando métodos de conversão de tipos, métodos chamando métodos do mesmo objecto, operador % que retorna o resto da divisão de dois inteiros.

Exemplo 12 :
```
 //header file para o TAD fracção.
```

```
//File easyfra.h

long mdc(long n,long d); //maximo divisor comum metodo de Euclides.

class fraccao {

private:

        long num; //numerador
        long den; //denominador

public:

        fraccao(long t,long m); //constructor comum
        void simplifica(void); //divisão pelo mdc
        ~fraccao() { /* nao faz nada*/ } //Não é preciso fazer nada.

        //operações matemáticas básicas
        fraccao soma (fracao j);
        fracao multiplicacao(fracao j);

        //operações de comparação
        int igual(fraccao t);
        int diferente(fraccao t);
        int maiorouigual(fracao t);

        //operações de input output
        void mostra(void); //exibe fracção no video
        void cria(void); //pergunta ao utilizador o valor da fracção

        //operações de conversão de tipos
        double convertedbl(void); //converte para double
        long convertelng(void); //converte para long
};

//implementação para a classe fracção.

#include <iostream.h
#include "easyfra.h"
```

```
long mdc(long n,long d) //maximo divisor comum

//metodo de Euclides +- 300 anos AC.
{
	if (n<0) n=-n;
	if (d<0) d=-d;
	while (d!=0) {
		long r=n % d; //%=MOD=Resto da divisao inteira.
		n=d;
		d=r;
}

return n;
}

void fraccao::simplifica(void)
{
	long commd;
	commd=mdc(num,den); //divisor comum
	num=num/commd;
	den=den/commd;
	if (den<0) { den=-den; num=-num;}; //move sinal para cima
}

fraccao::fraccao(long t,long m)
{
	num=(t);
	den=(m);
	simplifica(); //chamada para o mesmo objecto.
}

fraccao fraccao::soma(fraccao j)
{
	fraccao g((num*j.den)+(j.num*den),den*j.den);
	return g;
}

fraccao fraccao::multiplicacao(fraccao j)
```

```
{
	fraccao g(num*j.num,den*j.den);
	return g;
}

int fraccao::igual(fraccao t)
{
	return ((num*t.den)==(den*t.num));
	//funccciona bem mesmo para não simplificada
}

int fraccao::diferente(fraccao t)
{
	return ((num*t.den)!=(den*t.num));
}

int fraccao::maiorouigual(fraccao t)
{
	return ((num*t.den)=(t.num*den));
}

void fraccao::mostra(void)
{
	cout << "(" << num << "/" << den << ")";
}

void fraccao::cria(void)
{
	cout << "Numerador:";
	cin num;
	cout << "Denominador:";
	cin den;
	simplifica();
}

double fraccao::convertedbl(void)
{
	double dbl;
```

```
        dbl=(double(num)/double(den));
        return dbl;
}

long fracao::convertelng(void) //conversao para long
{
        long lng;

        lng=num/den;
        return lng;
}

#include <iostream.h
#include "easyfra.h" //nossa definicao da classe
#include <stdio.h

main()
{
        fraccao a(0,1),b(0,1);

        cout << " Entre com fracção a: ";
        a.cria();
        a.mostra();
        cout << " Entre com fracção b: ";
        b.cria();
        b.mostra();

        fraccao c(a.soma(b)); //c(a+b)
        cout << endl << "c de a+b:";
        c.mostra();
        cout << endl << "a*b";
        c=a.multiplicacao(b);
        c.mostra();
        cout << endl << "a+b";
        c=a.soma(b);
        c.mostra();
        cout << endl << "a=b";
        cout << a.maiorouigual(b);
        cout << endl << "a==b";
```

```
        cout << a.igual(b);
        cout << endl << "a!=b";
        cout << a.diferente(b);
        cout << endl << "long(a) ";
        cout << a.convertelng();
        cout << endl << "double(a) ";
        cout << a.convertedbl();
        return 0;
}
```

Comentários: Observa o seguinte código usado no programa: fraccao c(a.soma(b));. O resultado de a.soma(b) é uma fracção, mas não criamos um constructor que recebe uma fracção como argumento, como isso foi possível no programa?

Simples, a linguagem oferece para as classes que você cria a cópia bit a bit. Cuidado com o uso dessa cópia em conjunto com alocação dinâmica, objectos poderão ter cópias iguais de ponteiros, ou seja compartilhar uso de posições na memória. De qualquer forma, em 0 explicaremos como redefinir esta cópia de modo a evitar situações indesejáveis como a cópia de ponteiros e explicaremos melhor os perigos de usar este tipo de cópia em conjunto com alocação dinâmica.

Resultado do programa: Entre com fracção a: Numerador:4
Denominador:2

(2/1) Entre com fracção b: Numerador:5
Denominador:3

(5/3)
c de a+b:(11/3)
a*b(10/3)
a+b(11/3)
a=b1
a==b0
a!=b1
long(a) 2
double(a) 2