PUC-Minas - Ciência da Computação
AED1 – Estudo Dirigido 13

Tema: Introdução à programação V
Atividade: Grupos de dados heterogêneos - Classes

01.) Editar e salvar um esboço de classe em C++, cujo nome será Erro.hpp, que conterá definições para tratamento de erro, em uso posterior:

```
/**
 * Classe para tratar erro.
 */

#ifndef  _ERRO_H_
#define _ERRO_H_

class Erro
{
  /*
   * tratamento de erro.
     Codigos de erro:
     0. Nao ha' erro.
   */

  /**
   * atributos privados.
   */
   private:
     int erro;

   protected:

  // ---------------------------------------- metodos para acesso restrito

  /**
   * Metodo para estabelecer novo codigo de erro.
     @param codigo de erro a ser guardado
   */
   void setErro ( int codigo )
   {
      erro = codigo;
   } // end setErro ( )
```

```
/**
 * definicoes publicas.
 */
public:
/**
  * Destrutor.
  */
 ~Erro ( )
  { }

 /**
  * Construtor padrao.
  */
  Erro ( )
  {
   // atribuir valor inicial
     erro = 0;
  } // end constructor (padrão)

// ---------------------------------------- metodos para acesso

/**
 * Funcao para obter o codigo de erro.
   @return codigo de erro guardado
 */
 int getErro ( )
 {
    return ( erro );
 } // end getErro ( )

}; // end class Erro

#endif
```

OBS.:
Notar o uso do ponto-e-vírgula (';') após a definição.
A restrição ao acesso estará vinculada à derivação entre classes, conforme se fará a seguir.

Editar e salvar outro esboço de classe em C++, na mesma pasta, cujo nome será Contato.hpp, que conterá definições sobre dados de uma pessoa: nome e telefone.

```
/*
  Contato.hpp  - v0.0. - __ / __ / _____
  Author: _____________________

 */

// ------------------------------------------------ definicoes globais

#ifndef  _CONTATO_H_
#define _CONTATO_H_

// dependencias

#include <iostream>
using std::cin  ;          // para entrada
using std::cout;           // para saida
using std::endl;           // para mudar de linha

#include <iomanip>
using std::setw;           // para definir espacamento

#include <string>
using std::string;         // para cadeia de caracteres

#include <fstream>
using std::ofstream;    // para gravar arquivo
using std::ifstream;    // para ler    arquivo


// outras dependencias

void pause ( std::string text )
{
   std::string dummy;
   std::cin.clear ( );
   std::cout << std::endl << text;
   std::cin.ignore( );
   std::getline(std::cin, dummy);
   std::cout << std::endl << std::endl;
} // end pause ( )
```

```
#include "Erro.hpp"

// ------------------------------------------------  definicao de classe

/**
 * Classe para tratar contatos, derivada da classe Erro.
 */
class Contato : public Erro
{
 /**
  * atributos privados.
  */
  private:
    string nome;
    string fone;

 /**
  * definicoes publicas.
  */
  public:
  /**
   * Destrutor.
   */
  ~Contato ( )
   { }

  /**
   * Construtor padrao.
   */
   Contato ( )
   {
    // atribuir valores iniciais vazios
       nome = "";
       fone   = "";
   } // end constructor (padrao)

}; // fim da classe Contato

using ref_Contato = Contato*;  // similar a typedef Contato* ref_Contato;

#endif
```

OBS.:
Notar, mais uma vez, o encerramento da definição da classe com (';').
A última definição (***type alias***) irá auxiliar nas definições de referências para objetos dessa classe.

Editar outro programa em C++, na mesma pasta, cujo nome será Exemplo1300.cpp, para testar definições da classe Contato:

```
/*
  Exemplo1300 - v0.0. - __ / __ / _____
  Author: _____

 */

// ---------------------------------------------- classes

#include "Contato.hpp"  // classe para tratar dados de pessoas

// ---------------------------------------------- definicoes globais

using namespace std;

// ---------------------------------------------- metodos

/**
  Method_00 - nao faz nada.
 */
void method_00 ( )
{
 // nao faz nada
} // end method_00 ( )

/**
  Method_01 - Testar definicoes da classe.
 */
void method_01 ( )
{
 // definir dados
   Contato       pessoa1;
   ref_Contato pessoa2 = nullptr;
   ref_Contato pessoa3 = new Contato ( );

 // identificar
   cout << "\nMethod_01 - v0.0\n" << endl;

 // encerrar
   pause ( "Apertar ENTER para continuar" );
} // end method_01 ( )
```

```
// ------------------------------------------------- acao principal

/*
  Funcao principal.
  @return codigo de encerramento
*/
int main ( int argc, char** argv )
{
 // definir dado
    int x = 0;              // definir variavel com valor inicial

 // repetir até desejar parar
    do
    {
     // identificar
        cout << "EXEMPLO1300 - Programa - v0.0\n           " << endl;

     // mostrar opcoes
        cout << "Opcoes                                    " << endl;
        cout << " 0 - parar                                " << endl;
        cout << " 1 - testar definicao de contatos (objetos) " << endl;

     // ler do teclado
        cout << endl << "Entrar com uma opcao: ";
        cin  >> x;

     // escolher acao
        switch ( x )
        {
          case 0:
            method_00 ( );
           break;
          case 1:
             method_01 ( );
           break;
          default:
            cout << endl << "ERRO: Valor invalido." << endl;
        } // end switch
    }
    while ( x != 0 );

 // encerrar
    pause  ( "Apertar ENTER para terminar" );
    return ( 0 );
} // end main ( )
```

As referências para objetos da classe receberão valores iniciais definidos pelo construtor padrão.
A reciclagem do espaço será feita automaticamente de acordo com à definição do destrutor.

02.) Compilar o programa.
Se houver erros, resolvê-los e compilar novamente, até que todos tenham sido resolvidos.
Se não houver erros, seguir para o próximo passo.
Em caso de dúvidas, consultar a apostila, recorrer aos monitores ou apresentá-las ao professor.

03.) Executar o programa. Observar as saídas. Registrar os dados e os resultados.

04.) Acrescentar à definição da classe Contato os métodos abaixo:

```
// ------------------------------------ metodos para acesso

  /**
   * Metodo para atribuir nome.
   * @param nome a ser atribuido
   */
   void setNome ( std::string nome )
   {
     this->nome = nome;
   } // end setNome ( )

  /**
   * Metodo para atribuir telefone.
   * @param fone a ser atribuido
   */
   void setFone ( std::string fone )
   {
     this->fone = fone;
   } // end setFone ( )

  /**
   * Funcao para obter nome.
   * @return nome armazenado
   */
   std::string getNome ( )
   {
     return ( this->nome );
   } // end getNome ( )

  /**
   * Funcao para obter fone.
   * @return fone armazenado
   */
   std::string getFone ( )
   {
     return ( this->fone );
   } // end getFone ( )
```

Na parte principal do programa, incluir a chamada do método para testar as novas definições.

```
/**
  Method_02 - Testar atribuicoes.
 */
void method_02 ( )
{
 // definir dados
    Contato       pessoa1;
    ref_Contato pessoa2 = nullptr;
    ref_Contato pessoa3 = new Contato ( );

 // identificar
    cout << "\nMethod_02 - v0.0\n" << endl;

 // testar atribuicoes
    pessoa1.setNome   ( "Pessoa_01" );
    pessoa1.setFone    ( "111" );
    pessoa3->setNome ( "Pessoa_03" );
    pessoa3->setFone  ( "333" );

    cout << "pessoa1 - { " << pessoa1.getNome   ( ) << ", " << pessoa1.getFone    ( ) << " }" << endl;
    cout << "pessoa3 - { " << pessoa3->getNome ( ) << ", " << pessoa3->getFone ( ) << " }" << endl;

 // encerrar
    pause ( "Apertar ENTER para continuar" );
} // end method_02 ( )
```

05.) Acrescentar à definição da classe Contato o método abaixo:

```
    /**
     * Funcao para obter dados de uma pessoa.
     * @return dados de uma pessoa
     */
     std::string toString ( )
     {
       return ( "{ "+getNome( )+", "+getFone( )+" }" );
     } // end toString ( )
```

Na parte principal do programa, incluir a chamada do método para testar o novo.

```
/**
  Method_03 - Testar recuperacao de dados.
 */
void method_03 ( )
{
 // definir dados
    Contato      pessoa1;
    ref_Contato pessoa2 = nullptr;
    ref_Contato pessoa3 = new Contato ( );

 // identificar
    cout << "\nMethod_03 - v0.0\n" << endl;

 // testar atribuicoes
    pessoa1.setNome   ( "Pessoa_01" );
    pessoa1.setFone    ( "111" );
    pessoa3->setNome ( "Pessoa_03" );
    pessoa3->setFone  ( "333" );

    cout << "pessoa1 - " << pessoa1.toString   ( ) << endl;
    cout << "pessoa3 - " << pessoa3->toString ( ) << endl;

 // encerrar
    pause ( "Apertar ENTER para continuar" );
} // end method_03 ( )
```

06.) Acrescentar novo construtor à classe para criar objeto com valores iniciais.

```
    /**
     * Construtor alternativo.
     * @param nome_inicial a ser atribuido
     * @param fone_inicial   a ser atribuido
     */
     Contato ( std::string nome_inicial, std::string fone_inicial )
     {
      // atribuir valores iniciais

        nome = nome_inicial;
        fone   =  fone_inicial;

     } // end constructor (alternativo)
```

OBS.:
Notar que as atribuições não verificam as validades do que estiver sendo armazenado.

Na parte principal, acrescentar um método para testes.

```
/**
  Method_04 - Testar construtor alternativo.
 */
void method_04 ( )
{
 // definir dados
    Contato       pessoa1 ( "Pessoa_01", "111" );
    ref_Contato    pessoa2 = nullptr;
    ref_Contato pessoa3 = new Contato ( "Pessoa_03", "333" );

 // identificar
    cout << "\nMethod_04 - v0.0\n" << endl;

 // testar atribuicoes

    cout << "pessoa1 - " << pessoa1.toString   ( ) << endl;
    cout << "pessoa3 - " << pessoa3->toString ( ) << endl;

 // encerrar
    pause ( "Apertar ENTER para continuar" );
} // end method_04 ( )
```

07.) Copiar a versão atual do programa para outra nova – Exemplo1305.cpp.

08.) Acrescentar tratamento de erros à classe Contato.
Incluir nos construtores atribuições de código inicial para erro, conforme exemplo abaixo:

```
    /**
      * Construtor padrao.
      */
      Contato ( )
      {
         setErro ( 0 );   // nenhum erro, ainda

       // atribuir valores iniciais vazios
         nome = "";
         fone   = "";
      } // end constructor (padrão)
```

Rever os métodos para acesso para incluir o tratamento de erros.

```
/**
 * Metodo para atribuir nome.
 * @param nome a ser atribuido
 */
 void setNome ( std::string nome )
 {
   if ( nome.empty ( ) )
      setErro ( 1 ); // nome invalido
   else
      this->nome = nome;
 } // end setNome ( )

/**
 * Metodo para atribuir telefone.
 * @param fone a ser atribuido
 */
 void setFone ( std::string fone )
 {
   if ( fone.empty ( ) )
      setErro ( 2 ); // fone invalido
   else
      this->fone = fone;
 } // end setFone ( )


/**
 * Construtor alternativo.
 * @param nome_inicial a ser atribuido
 * @param fone_inicial   a ser atribuido
 */
 Contato ( std::string nome_inicial, std::string fone_inicial )
 {
     setErro ( 0 );   // nenhum erro, ainda

  // atribuir valores iniciais

     setNome ( nome_inicial ); // nome = nome_inicial;
     setFone ( fone_inicial     ); // fone   =  fone_inicial;


 } // end constructor (alternativo)
```

Na parte principal, acrescentar chamada a um método para testar o tratamento de erros.

```
/**
  Method_05 - Testar construtor alternativo.
 */
void method_05 ( )
{
 // definir dados
   Contato         pessoa1 ( "Pessoa_01", "111" );
   ref_Contato   pessoa2 = nullptr;
  ref_Contato pessoa3 = new Contato ( "", "333" );

 // identificar
   cout << "\nMethod_05 - v0.0\n" << endl;

 // testar atribuicoes

   cout << "pessoa1 - " << pessoa1.toString   ( ) << " (" << pessoa1.getErro( )   << ")" << endl;
   cout << "pessoa3 - " << pessoa3->toString ( ) << " (" << pessoa3->getErro( ) << ")" << endl;

 // encerrar
   pause ( "Apertar ENTER para continuar" );
} // end method_05 ( )
```

OBS.:
Notar que a obtenção do código de erro é possível, mas alterá-lo fora da classe, não.

09.) Compilar o programa novamente.
Se houver erros, resolvê-los e compilar novamente, até que todos tenham sido resolvidos.
Se não houver erros, seguir para o próximo passo.

10.) Executar o programa.
Observar as saídas.
Registrar os dados e os resultados.

11.) Acrescentar um método para indicar a existência de erro.

```
 /**
  * indicar a existencia de erro.
  */
 bool hasErro ( )
 {
   return ( getErro( ) != 0 );
 } // end hasErro ( )
```

Na parte principal, acrescentar um método para testar o tratamento de erro.

```
/**
  Method_06 - Testar construtor alternativo.
 */
void method_06 ( )
{
 // definir dados
   Contato      pessoa1 ( "Pessoa_01", "111" );
   ref_Contato pessoa2 = nullptr;
   ref_Contato pessoa3 = new Contato ( "", "333" );

 // identificar
   cout << "\nMethod_06 - v0.0\n" << endl;

 // testar atribuicoes

   if ( ! pessoa1.hasErro( ) )
     cout << "pessoa1 - " << pessoa1.toString( )  << endl;
   else
     cout << "pessoa1 tem erro (" << pessoa1.getErro( )  << ")" << endl;

   if ( ! pessoa3->hasErro( ) )
     cout << "pessoa3 - " << pessoa3->toString( ) << endl;
   else
     cout << "pessoa3 tem erro (" << pessoa3->getErro( ) << ")" << endl;

 // encerrar
   pause ( "Apertar ENTER para continuar" );
} // end method_06 ( )
```

OBS.:
O novo método facilitará os testes para verificação de possíveis erros.

12.) Compilar o programa novamente.
Se houver erros, resolvê-los e compilar novamente, até que todos tenham sido resolvidos.
Se não houver erros, seguir para o próximo passo.

13.) Executar o programa.
Observar as saídas.
Registrar os dados e os resultados.

14.) Na parte principal, acrescentar chamada a um método para testar o tratamento de erros.

```
/**
  Method_07 - Testar atribuicoes e tratamento de erro.
 */
void method_07 ( )
{
 // definir dados
   Contato       pessoa1 ( "Pessoa_01", "111" );
   ref_Contato pessoa2 = nullptr;
   ref_Contato pessoa3 = new Contato ( "", "333" );

 // identificar
   cout << "\nMethod_07 - v0.0\n" << endl;

 // testar atribuicoes

   pessoa2 = &pessoa1;        // copiar endereco de objeto
   if ( ! pessoa2->hasErro( ) )
     cout << "pessoa1 - " << pessoa2->toString( )  << endl;
   else
     cout << "pessoa1 tem erro (" << pessoa2->getErro( )  << ")" << endl;

   pessoa2 =  pessoa3;         // vincular-se a outro objeto
   if ( ! pessoa2->hasErro( ) )
     cout << "pessoa3 - " << pessoa2->toString( ) << endl;
   else
     cout << "pessoa3 tem erro (" << pessoa2->getErro( ) << ")" << endl;

 // encerrar
   pause ( "Apertar ENTER para continuar" );
} // end method_07 ( )
```

OBS.:
Notar que a referência irá indicar ambos os casos representados, um por vez.

15.) Acrescentar um construtor alternativo baseado em cópia.

```
   /**
    * Construtor alternativo baseado em copia.
    */
    Contato ( Contato const & another )
    {
     // atribuir valores iniciais por copia
        setErro ( 0 );                    // copiar erro
        setNome ( another.nome ); // copiar nome
        setFone ( another.fone    ); // copiar fone
    } // end constructor (alternativo)
```

Na parte principal, acrescentar um método para testes.

```
/**
  Method_08 - Testar atribuicoes e tratamento de erro.
 */
void method_08 ( )
{
 // definir dados
   Contato       pessoa1 ( "Pessoa_01", "111" );
   ref_Contato pessoa2 = nullptr;
   ref_Contato pessoa3 = new Contato ( "", "333" );
  ref_Contato pessoa4 = nullptr;

 // identificar
   cout << "\nEXEMPLO1308 - Method_08 - v0.0\n" << endl;

 // testar atribuicoes

   pessoa2 =  new Contato ( pessoa1 );
   if ( pessoa2 )
     cout << "pessoa2 - " << pessoa2->toString( ) << endl;
   else
     cout << "pessoa2 tem erro (" << pessoa2->getErro( ) << ")" << endl;

   if ( pessoa3 )  // o teste de existencia deve ser feito previamente
   {
     pessoa2 =  new Contato ( *pessoa3 );
     if ( pessoa2 )
      cout << "pessoa2 - " << pessoa2->toString( ) << endl;
     else
      cout << "pessoa2 tem erro (" << pessoa3->getErro( ) << ")" << endl;
   } // end if

   if ( pessoa4 )  // o teste de existencia deve ser feito previamente
   {
     pessoa2 =  new Contato ( *pessoa4 );
     if ( pessoa2 )
       cout << "pessoa2 - " << pessoa2->toString( ) << endl;
     else
       cout << "pessoa2 tem erro (" << pessoa4->getErro( ) << ")" << endl;
   } // end if

 // encerrar
   pause ( "Apertar ENTER para continuar" );
} // end method_08 ( )
```

OBS.:
Notar que ***todos*** os testes de existência deverão ser feitos ***previamente*** aos usos.

16.) Compilar o programa novamente.
Se houver erros, resolvê-los e compilar novamente, até que todos tenham sido resolvidos.
Se não houver erros, seguir para o próximo passo.

17.) Executar o programa.
Observar as saídas.
Registrar os dados e os resultados.

18.) Na parte principal, acrescentar um método para testar a atribuição de valores ao objeto.

```
/**
   Method_09 - Testar arranjo de objetos (1).
 */
void method_09 ( )
{
 // definir dados
    Contato pessoa [ 3 ];
    int x = 0;

 // identificar
    cout << "\nMethod_09 - v0.0\n" << endl;

 // testar atribuicoes

    pessoa [ 0 ].setNome ( "Pessoa_1" );
    pessoa [ 0 ].setFone ( "111" );

    pessoa [ 1 ].setNome ( "Pessoa_2" );
    pessoa [ 1 ].setFone ( "222" );

    pessoa [ 2 ].setNome ( "Pessoa_3" );
    pessoa [ 2 ].setFone ( "333" );

    for ( x=0; x < 3; x=x+1 )
    {
      cout << x << " : " << pessoa[ x ].toString( ) << endl;
    } // end for

 // encerrar
    pause ( "Apertar ENTER para continuar" );
} // end method_09 ( )
```

19.) Compilar o programa novamente.
Se houver erros, resolvê-los e compilar novamente, até que todos tenham sido resolvidos.
Se não houver erros, seguir para o próximo passo.

20.) Executar o programa.
Observar as saídas.
Registrar os dados e os resultados.

21.) Na parte principal, acrescentar um método para testar a atribuição de valores ao objeto.

```
/**
   Method_10 - Testar arranjo de referencias para objetos (2).
 */
void method_10 ( )
{
 // definir dados
    Contato *pessoa [ 3 ];
    int x = 0;

 // identificar
    cout << "\nMethod_10 - v0.0\n" << endl;

 // testar atribuicoes

    pessoa [ 0 ] = new Contato ( "Pessoa_1", "111" );

    pessoa [ 1 ] = new Contato ( "Pessoa_2", "222" );

    pessoa [ 2 ] = new Contato ( "Pessoa_3", "333" );

    for ( x=0; x < 3; x=x+1 )
    {
      cout << x << " : " << pessoa[ x ]->toString( ) << endl;
    } // end for

 // encerrar
    pause ( "Apertar ENTER para continuar" );
} // end method_10 ( )
```

OBS.:
Notar que ***todas*** as atribuições, agora, se valem de usos do construtor alternativo.

22.) Compilar o programa novamente.
Se houver erros, resolvê-los e compilar novamente, até que todos tenham sido resolvidos.
Se não houver erros, seguir para o próximo passo.

23.) Executar o programa.
Observar as saídas.
Registrar os dados e os resultados.

Exercícios:

DICAS GERAIS: Consultar o Anexo CPP 02 na apostila para outros exemplos.
NÃO usar métodos ou funções prontos em bibliotecas/classes nativas da linguagem C.
Prever, realizar e registrar todos os testes efetuados.
Integrar as chamadas de todos os programas em um só.
Restrições:
- Usar definições de dados e métodos em classes.
- Usar a classe **string** de C++.

01.) Incluir um método público à classe (1311) para
ler do teclado e atribuir um valor ao nome (atributo de certo objeto).
Incluir um método para testar essa nova característica.
DICA: Testar se o nome não está vazio.

Exemplo: contato1.readName (“Nome: ”);

02.) Incluir um método público à classe (1312) para
ler do teclado e atribuir um valor ao telefone (atributo de certo objeto).
Incluir um método para testar essa nova característica.
DICA: Testar se o telefone não está vazio.

Exemplo: contato1.readPhone (“Fone: ”);

03.) Incluir um método privado à classe (1313) para
testar se o valor de um telefone é válido, ou não.
Incluir um método para testar essa nova característica.
DICA: Testar se as posições contêm apenas algarismos e o símbolo ‘-‘.

Exemplo: bool OK = contato1.isValidPhone (“9999-9999”);

04.) Incluir um método público à classe (1314) para
ler dados de arquivo, dado o nome do mesmo, e armazenar em um objeto dessa classe.
Incluir um método para testar essa nova característica.
DICA: Se o nome não estiver vazio, confirmar a alteração.

Exemplo: contato1.readFromFile (“Pessoa1.txt”);

05.) Incluir um método público à classe (1315) para
gravar dados de uma pessoa em arquivo, dado o nome do mesmo.
Incluir um método para testar essa nova característica.
DICA: Gravar o tamanho também do arquivo, primeiro, antes dos outros dados.

Exemplo: contato.writeToFile (“Pessoa1.txt”);

06.) Incluir um novo atributo à classe (1316) para
um segundo telefone e modificar os construtores para lidar com mais essa possibilidade.
Incluir um método para testar essa nova característica.
DICA: Incluir um item para guardar a quantidade de telefones e
trocar armazenador para arranjo.
Recomendável usar a classe arranjo definida anteriormente.

Exemplo: contato1 = new Contato (“nome1”, “99999-1111”, “98888-2222”);

07.) Incluir um novo atributo à classe (1317) para
indicar quantos telefones estão associados a cada objeto.
Incluir um método para obter essa informação.
Incluir um método para testar essa nova característica.
DICA: Usar as modificações sugeridas no exemplo anterior.

Exemplo: int n = contato1.phones ( );

08.) Incluir um método público (1318) para
para atribuir valor ao segundo telefone.
Incluir um método para testar essa nova característica.
DICA: Se o contato só tiver um telefone, perguntar se quer acrescentar mais um número,
e mudar automaticamente a quantidade deles, se assim for desejado.

Exemplo: contato.setPhone2a (“97777-3333”);

09.) Incluir um método público (01319) para
alterar o valor apenas do segundo telefone.
Incluir um método para testar essa nova característica.
DICA: Se o contato não tiver dois telefones, uma situação de erro deverá ser indicada.

Exemplo: contato.setPhone2b (“97777-3333”);

10.) Incluir um método público (01320) para
remover apenas o valor do segundo telefone.
Incluir um método para testar essa nova característica.
DICA: Se o contato só tiver um telefone, uma situação de erro deverá ser indicada.

Exemplo: contato.setPhone2c ("");

Tarefas extras

E1.) Fazer modificações na classe Contato (013E1)
para lidar com qualquer quantidade de telefones, menor ou igual a 10.
Incluir testes para essa nova característica.
DICA: Guardar a quantidade de telefones e, separadamente, os telefones em arranjo.

E2.) Fazer modificações na classe Contato (013E2)
para lidar também com endereços (residencial e profissional).
Incluir testes para essa nova característica.
DICA: Guardar separadamente o endereço residencial e o profissional.
Experimentar definir classe com os dados e usar outra classe derivada da primeira.