```
//---------------------------------------dla-------------------------------------------
            /*
  Nomefile | Matrice.h
           |
  Tipofile | Header C++
           | Ultima revisione: 05/01/2009
           |
     Autore| Angelo De Luna - C\da Oppidi-Varano, 68 -84022- Campagna (SA)
           | Sito Internet: http://www.angelodeluna.it
           | E_Mail: angelo.de.luna@alice.it
           | tel.: 3314982802
           |
     Scopo | Dichiarazione e definizione della classe template DLA::Matrice<T>
           | per la gestione delle matrici.
           | Trattandosi di una classe template si sono implementati i metodi
           | nello stesso file (header) della dichiarazione poiché non tutti i
           | compilatori supportano le classi template nel formato
           | "Intestazione-Sorgente".
           | La classe è realizzata come template in modo da rendere possibile
           | l'utilizzo della stessa per qualsiasi tipo di dato. La
           | specializzazione per un tipo di dato sarà fatta dal compilatore
           | durante la compilazione stessa.
           */
//-------------------------------------------------------------------------------------
#ifndef MatriceH // direttiva di controllo al preprocessore
#define MatriceH // direttiva di definizione al preprocessore
//-------------------------------------------------------------------------------------
#include <mem.h> // richiesto da memcpy(), memset() e memcmp()
#pragma hdrstop // termina la ricerca dei file header nella libreria standard
//-------------------------------------------------------------------------------------
using namespace std; // usa namespace standard per gli oggetti invocati
//-------------------------------------------------------------------------------------
// Definizione mio namespace
namespace Angelo_De_Luna_07_03_66
{
   template<typename T> class Matrice
   {
      protected:
         T* punt; // puntatore ad oggetto T
         int size; // dimensione in byte della matrice
         int m_rows; // numero righe
         int m_cols; // numero colonne
         void setEl(T* dest, const T* value); // copia di elemento
      public:
         Matrice(); // costruttore predefinito
         Matrice(int rows, int cols); // costruttore parametrizzato
         Matrice(const Matrice<T>& m); // costruttore di copia
         ~Matrice(); // distruttore
         void resize(int rows, int cols); // allocazione dinamica della memoria
         T* operator[](int row); // primo operatore []
         T& operator[](int col) const; // secondo operatore []
         int rows() const; // restituzione numero righe
         int cols() const; // restituzione numero colonne
         void reset(); // azzeramento celle
         bool operator==(const T valore) const; // operatore == (ugua.tra elem.)
         bool operator!=(const T valore) const; // operatore != (disu.tra elem.)
         bool operator==(const Matrice<T>& m) const; // operatore == (ug.tra m.)
         bool operator!=(const Matrice<T>& m) const; // operatore != (di.tra m.)
         T& operator=(const T valore); // operatore = (assegnazione tra elemen.)
         Matrice<T>& operator=(const Matrice<T>& m); // operatore = (as.tra ma.)
   };
   //-------------------------------------------------------------------------------
```

```
template<typename T> Matrice<T> setEl(T* dest, const T* value)
{
   memcpy(dest,value,sizeof(T));
}

          /*
  Membri | Tutti i membri dati della classe sono definiti con visibilità
protetti | protetta onde permetterne l'ereditarietà in caso di derivazione.
         | Da notare che la variabile che conterrà la matrice è un array
         | monodimensionale allocato dinamicamente sull'heap che non fa parte
         | della classe, ma che è puntato dal campo privato punt della classe;
         | la matrice è dunque implementata come variabile semidinamica.
         | Seguono tre campi di tipo intero: size conterrà la dimensione in
         | byte della matrice, m_rows il numero delle righe e m_cols quello
         | delle colonne.
         | Definiamo anche una funzione di servizio, setEl(), sempre di
         | visibilità protetta che sarà utilizzata da altri metodi pubblici
         | che andremo a definire più avanti, e che consente la modifica del
         | valore di una singola cella. Si fa uso, qui, dalla funzione di
         | libreria standard memcpy(), la quale esegue una copia di basso
         | livello, bit per bit, nella cella puntata da "dest" del valore
         | puntato da "value".
         */
//---------------------------------------------------------------------------

// Costruttore predefinito
template<typename T> Matrice<T>::Matrice()
{
   punt=NULL;
   size=0;
   m_cols=m_rows=0;
}

// Costruttore parametrizzato
template<typename T> Matrice<T>::Matrice(int rows, int cols)
{
   punt=NULL;
   resize(rows,cols);
   reset();
}

// Costruttore di copia
template<typename T> Matrice<T>::Matrice(const Matrice<T>& m)
{
   punt=NULL;
   resize(m.m_rows,m.m_cols);
   memcpy(punt,m.punt,size);
}

// Distruttore
template<typename T> Matrice<T>::~Matrice()
{
   if(punt!=NULL) delete[] punt;
}

          /*
Costrut-| Oltre naturalmente al costruttore predefinito e al distruttore,
   tori | sono previsti altri due metodi.
      e | Il primo di essi è un costruttore parametrizzato per la creazione
distrut-| di una matrice di "rows" righe e "cols" colonne mediante il metodo
   tore | resize(), ed inizializzata a NULL tramite il metodo reset().
```

```
         | Il secondo è il costruttore di copia il quale crea una matrice,
         | mediante il metodo resize(), di dimensione pari a quella della
         | matrice il cui reference è passato come parametro esplicito ed
         | inizializzata ai valori di quest'ultima tramite la funzione di
         | libreria standard memcpy().
         | Si ricordi che il costruttore di copia è invocato automaticamente
         | quando un oggetto ne inizializza un altro, sia esplicitamente
         | mediante un'istruzione dichiarativa con relativa inizializzazione,
         | sia in modo implicito come avviene nel caso di passaggio di oggetti
         | a funzioni come parametri valore o nel caso di creazione di oggetti
         | temporanei da utilizzare come valore restituito da una funzione. In
         | assenza del costruttore di copia, in tutti i casi suddetti, il
         | compilatore predisporrà per default una copia bit per bit dello
         | oggetto, ma ciò, nel nostro caso, creerebbe seri problemi quando lo
         | oggetto temporaneo sarà cancellato, perchè fuori scopo, in quanto
         | il distruttore libererà della memoria puntata da punt a cui ancora
         | può riferirsi l'oggetto originario.
         | Ricordiamo anche che il costruttore di copia non viene invece
         | invocato durante le istruzioni di assegnazione.
         */
//---------------------------------------------------------------------------

// Allocazione dinamica della memoria
template<typename T> void Matrice<T>::resize(int rows, int cols)
{
   if(rows<0) throw" NUMERO DI RIGHE NON VALIDO!";
   if(cols<0) throw" NUMERO DI COLONNE NON VALIDO!";
   size=sizeof(T)*rows*cols;
   if(size==0)
   {
      if(punt!=NULL) delete[] punt;
      punt=NULL;
   }
   else
   {
      if(punt!=NULL) delete[] punt;
      punt=new T[size];
      if(punt==NULL) throw" MEMORIA INSUFFICIENTE!";
   }
   m_cols=cols;
   m_rows=rows;
}

          /*
 Alloca-  | Il metodo resize() è il motore della classe. Esso oltre ad essere
   zione  | utilizzato direttamente dall'operatore di assegnazione = tra
dinamica | matrici, dal costruttore di copia e dal costruttore parametrizzato,
  della   | può sempre essere invocato esplicitamente ed indipendentemente in
memoria   | qualità di funzione membro pubblico.
          | Attraverso questo metodo dimensioniamo e/o ridimensioniamo la
          | matrice.
          | La variabile punt è un puntatore, quindi se esso punta ad un blocco
          | di memoria allocato in precedenza, e quindi ha un valore diverso da
          | NULL, per prima cosa rilasciamo la memoria con delete, e poi, se lo
          | spazio richiesto, size, non è nullo, la riallochiamo, o la
          | allochiamo per la prima volta, con new, altrimenti impostiamo punt
          | a NULL senza allocarne dell'altra.
          | Inoltre, se la memoria libera non è sufficiente a soddisfare la
          | richiesta, new farà puntare automaticamente la nostra variabile al
          | valore NULL. In questa sfortunata situazione lanciamo un'eccezione.
          | Altre due eccezioni saranno lanciate non appena si tenta di passare
          | un numero di righe o di colonne minore di zero.
```

```
            | In ogni caso, alla fine, aggiorniamo il contenuto dei campi dati,
            | m_rows e m_cols, per mezzo dei parametri espliciti.
            */
   //-------------------------------------------------------------------------

   // Primo operatore [] (indice riga)
   template<typename T> T* Matrice<T>::operator[](int row)
   {
      return punt+row*m_cols;
   }

   // Secondo operatore [] (indice colonna)
   template<typename T> T& Matrice<T>::operator[](int col) const
   {
      return *(this+col);
   }

             /*
 Operatori | Il metodo di memorizzazione delle celle è molto semplice ed
        di | efficiente. Le righe sono salvate l'una di seguito all'altra.
 indicizza-| Per ottenere il valore alla cella di coordinate (i,j) saltiamo le
     zione | prime "i" righe (ricordiamo che gli indici i e j partono entrambi
            | da zero) e restituiamo il valore di seconda coordinata j.
            | L'operazione di indicizzazione è realizzata utilizzando due
            | operatori [] in successione. Infatti, applicando il primo operatore
            | [] ad un oggetto della classe si ottiene un puntatore (temporaneo)
            | al primo elemento della riga scelta il quale è passato come
            | argomento implicito al secondo operatore [] che restituisce il
            | reference dell'elemento richiesto.
            | Grazie a questo potente overloading, dunque, accederemo ad ogni
            | cella attraverso l'utilizzo della doppia parentesi quadra [][]
            | ancora una volta.
            */
   //-------------------------------------------------------------------------

   // Restituzione numero righe
   template<typename T> int Matrice<T>::rows() const
   {
      return m_rows;
   }

   // Restituzione numero colonne
   template<typename T> int Matrice<T>::cols() const
   {
      return m_cols;
   }

   // Azzeramento celle
   template<typename T> void Matrice<T>::reset()
   {
      memset(punt,NULL,size);
   }

             /*
     Altre | A questo punto aggiungiamo la possibilità di ottenere il numero di
operazioni | righe e colonne della matrice, rispettivamente metodi rows() e
            | cols().
            | Per impostare tutte le celle della matrice al valore NULL, come
            | di fatto avviene quando si invoca il costruttore parametrizzato,
            | possiamo usare il metodo reset() il quale richiama la funzione di
            | libreria standard memset(). Essendo di visibilità pubblica, il
            | metodo può anche essere invocato esplicitamente, dall'esterno, per
```

```
          | ogni oggetto creato.
          */
  //-------------------------------------------------------------------------

  // Operatore == (uguaglianza tra elementi)
  template<typename T> bool Matrice<T>::operator==(const T valore) const
  {
     T* value=&valore;
     if(memcmp(punt,value,sizeof(T))==0) return true;
     return false;
  }

  // Operatore != (disuguaglianza tra elementi)
  template<typename T> bool Matrice<T>::operator!=(const T valore) const
  {
     return !(*(punt)==valore);
  }

           /*
Operatori | Seguono le implementazioni dell'overloading dei due operatori
 booleani | logici relazionali tra elementi, grazie ai quali possiamo sapere
      tra | se due elementi sono uguali oppure no, o se un elemento è uguale
 elementi | oppure no al valore di una qualsiasi variabile di tipo T.
          | Si presti particolare attenzione al fatto che il secondo operatore
          | è implementato facendo uso del primo e che la funzione di libreria
          | standard memcmp() esegue un confronto bit per bit dei blocchi di
          | memoria puntati dai primi due parametri (punt e value) restituendo
          | il valore zero in caso di uguaglianza.
          | Precisiamo espressamente che, qui ed altrove (come per l'operatore
          | di assegnazione tra elementi), il puntatore punt all'atto della
          | chiamata del metodo (per esempio A[i][j]==valore, dove "valore" può
          | essere a sua volta, ma non necessariamente, un altro elemento
          | indicizzato del tipo B[h][k]) sta puntando all'elemento indicizzato
          | dall'operatore [][] posposto al nome, A, della matrice con cui lo
          | si invoca, perché quest'ultimo operatore ha la precedenza su quello
          | relazionale ==. Nelle chiamate di operatori tra elementi, pertanto,
          | il puntatore punt non necessariamente farà riferimento al primo
          | elemento della matrice, come invece avviene per gli omologhi
          | operatori tra matrici che definiremo appresso. Per lo stesso motivo
          | il terzo parametro di memcmp() è la dimensione in byte del tipo di
          | elemento, sizeof(T), e non la dimensione dell'intero blocco di
          | memoria, size, relativo alla matrice nella sua totalità.
          | Osserviamo, infine che, per mezzo della funzionalità di basso
          | livello, memcmp(), è possibile applicare gli operatori logici testé
          | implementati ad elementi di qualsiasi tipo, per i quali, quindi, la
          | uguaglianza e la disuguaglianza possono anche non essere definite a
          | priori.
          */
  //-------------------------------------------------------------------------

  // Operatore == (uguaglianza tra matrici)
  template<typename T> bool Matrice<T>::operator==(const Matrice<T>& m) const
  {
     if(m_rows!=m.m_rows) return false;
     if(m_cols!=m.m_cols) return false;
     if(size==m.size)
     {
        if(memcmp(punt,m.punt,size)==0) return true;
     }
     return false;
  }
```

```
  // Operatore != (disuguaglianza tra matrici)
  template<typename T> bool Matrice<T>::operator!=(const Matrice<T>& m) const
  {
     return !(*this==m);
  }

           /*
Operatori | In modo analogo, le implementazioni dell'overloading dei due
 booleani | operatori logici relazionali tra matrici permettono di verificare
      tra | se due matrici sono uguali oppure no (es.: A==B o A!=B).
  matrici | Anche qui il secondo operatore è implementato facendo uso del
          | primo e la funzione di libreria standard memcmp() di basso livello
          | rende possibile l'applicazione dei due operatori a matrici di
          | qualsiasi tipo.
          */
  //-------------------------------------------------------------------------

  // Operatore = (assegnazione tra elementi)
  template<typename T> T& Matrice<T>::operator=(const T valore)
  {
     T* value=&valore;
     setEl(punt,value);
     return punt;
  }

           /*
Operatore | Gli overloading dell'operatore di assegnazione = (come per altro
       di | quello dell'operatore [][] già visto) devono essere implementati
  assegna-| necessariamente per mezzo di funzioni membro.
    zione | L'assegnazione tra/di elementi è implementata facendo uso del
      tra | metodo protetto setEl() onde poter eseguire l'operazione anche nel
 elementi | caso in cui il tipo di dato T non la prevede. Si noti, tra l'altro,
          | che l'operatore restituisce il reference del valore puntato da punt
          | (elemento a destra del simbolo di =) proprio per consentire anche
          | assegnazioni concatenate, del tipo A[i][j]=B[h,k]=C[r][s]=...
          | ...=valore. A tal proposito ricordiamo che l'ordine con cui viene
          | interpretata la catena di assegnazioni, in assenza di parentesi
          | tonde, non è diverso da quello dell'operatore = predefinito, ovvero
          | da destra verso sinistra.
          */
  //-------------------------------------------------------------------------

  // Operatore = (assegnazione tra matrici)
  template<typename T> Matrice<T>& Matrice<T>::operator=(const Matrice<T>& m)
  {
     if(this!=&m)
     {
        resize(m.m_rows,m.m_cols);
        memcpy(punt,m.punt,size);
     }
     return *this;
  }

           /*
Operatore | Per prima cosa, con resize(), ridimensioniamo la matrice che deve
       di | ospitare i valori dell'assegnazione (matrice a sinistra del simbolo
  assegna-| di =) sulle dimensioni (m.m_rows e m.m_cols) di quella che
    zione | costituisce l'oggetto dell'assegnazione (matrice a destra del
      tra | simbolo di =); poi, per mezzo della funzione di libreria standard
  matrici | memcpy(), eseguiamo una copia di basso livello (bit per bit) del
          | blocco di memoria degli elementi di quest'ultima (puntato cioè da
          | m.punt) nel blocco di memoria degli elementi dalla prima (puntato
```

```
            | quindi da punt e uniformato alla stessa dimensione, size,
            | indipendentemente dalla sua dimensione originaria).
            | Si noti che, analogamente a quanto avviene nell'assegnazione di
            | elementi (ma non identicamente), la funzione operatore = tra
            | matrici restituisce il reference dell'oggetto puntato da this; ciò
            | permette, pure in questo caso, di utilizzare il simbolo di = anche
            | in una catena di assegnazioni tra matrici, per esempio A=B=C... ,
            | sempre sotto le stesse regole d'ordine di esecuzione viste in
            | precedenza.
            | Rispetto all'assegnazione tra/di elementi, l'assegnazione tra/di
            | matrici, a causa dell'attivazione del meccanismo di allocazione
            | dinamica della memoria, presenta delle complicazioni tecniche che
            | (dato il carattere esplicativo dei presenti commenti) è bene
            | trattare dettagliatamente.
            | La restituzione dei reference degli oggetti assegnati, e non degli
            | oggetti veri e propri, da parte dell'operatore di assegnazione tra
            | matrici è, in tal caso, indispensabile, perché permette di iterare
            | l'assegnazione stessa senza dover creare oggetti temporanei che,
            | come è noto, oltre ad impegnare maggior tempo e risorse di
            | elaborazione, darebbe luogo al problema dei PUNTATORI CONDIVISI.
            | Ricordiamo, infatti che, come già visto, il costruttore di copia
            | non è invocato per l'operatore di assegnazione, quindi, in tal
            | caso, il compilatore predisporrà una copia bit per bit degli
            | oggetti che devono essere restituiti. Questo, però, produrrebbe
            | delle conseguenze potenzialmente dannose in quanto la variabile
            | punt dell'oggetto alla sinistra del simbolo di = riceverebbe lo
            | stesso indirizzo di memoria relativo al bolocco degli elementi
            | della matrice alla destra del simbolo di =. In conclusione, se non
            | si adoperassero i reference come tipo di restituzione, la
            | assegnazione tra matrici produrrebbe due o più oggetti distinti
            | che, però, puntano tutti alla medesima locazione di memoria.
            | Di più; vi è ancora un'ultima circostanza pericolosa da prendere in
            | seria considerazione: L'AUTO ASSEGNAZIONE DISTRUTTIVA.
            | Questa è un'assegnazione del tipo A=A (non affatto improbabile per
            | chi deve gestire matrici e tabelle in modo efficiente e senza
            | eccezioni di sorta) la quale, anche se in apparenza sembra del
            | tutto innocua, nasconde un'insidia tuttaltro che trascurabile, per
            | la cui neutralizzazione si rende necessaria l'istruzione di
            | controllo in entrata, if(this!=&m). Se non ci fosse questa
            | verifica, infatti, nel caso in cui il parametro implicito (matrice
            | alla sinistra di =, il cui indirizzo di memoria è depositato nel
            | puntatore this) coincidesse con quello esplicito (matrice alla
            | destra di = il cui indirizzo è invece fornito da &m), il metodo
            | resize() invocato dall'operatore medesimo rilascerebbe la locazione
            | di memoria il cui contenuto è proprio l'argomento della
            | assegnazione, prima di completare, con memcpy(), l'assegnazione
            | stessa. Si deve quindi prevedere esplicitamente che, in caso di
            | auto assegnazione, non occorre effettuare alcuna operazione in
            | memoria.
            */
   //---------------------------------------------------------------------------

} // chiusura mio namespace
//-----------------------------------------------------------------------------

// Alias namespace
namespace DLA=Angelo_De_Luna_07_03_66;

            /*
 Namespace  | L'abbreviazione del namespace permetterà di riferirsi a questa
            | classe dall'esterno semplicemente anteponendo al suo nome la sigla
            | DLA seguita dal doppio due punti, secondo la seguente semplice
```

```
            | sintassi: DLA::Matrice<T> (dove al posto del simbolo T va inserito
            | il nome di un qualsiasi tipo di dati).
            */
//-------------------------------------------------------------------------------

#endif // fine direttiva di controllo al preprocessore
//-------------------------------------C++---------------------------------------
```