Allocazione dinamica

- Quando?
  - Tutte le volte in cui i dati possono crescere in modo non prevedibile staticamente a tempo di sviluppo
    - Un array con dimensione fissata a compile-time non è sufficiente

  → **È necessario avere "più" controllo sull'allocazione di memoria**

  →**Allocazione della memoria "by need"**

`malloc()`

- Allocazione tramite la funzione:

```
void* malloc(size_t dim);
```

- Il valore di ritorno è un puntatore senza tipo (**`void*`**) → un indirizzo all'inizio di un'area di memoria la cui dimensione è definita dall'argomento della **`malloc()`**
- Il sistema **ricorda**, **per ogni singola allocazione**, quanta memoria è stata allocata
- **Attenzione al `void*`** Può essere trasformato tramite *cast* in qualsiasi tipo di puntatore → anche in un tipo la cui dimensione non corrisponde al tipo allocato → **PERICOLOSISSIMO!**

`free()`

- Deallocazione tramite la procedura:

```
void free(void* p);
```

- Il sistema sa quanta memoria deallocare per quel puntatore (**ricorda** la relativa malloc)
- Se la memoria non viene correttamente deallocata → ***memory leaking***
- In caso di strutture dati condivise, come si decide quando deallocare la memoria?
- In ogni momento occorre sapere **chi** ha in uso una certa struttura condivisa per ***deallocare solamente quando più nessuno ne ha un riferimento***

Garbage Collection

- Nei moderni linguaggi di programmazione, la deallocazione della memoria non è più un problema
- Esistono ***sistemi automatici di recupero della memoria allocata ma non più usata*** → allocazione esplicita, ***deallocazione automatica***
- Il sistema sa sempre quanti e quali puntatori puntano ad una certa area di memoria → quando un'area di memoria non è più puntata da nessuno, viene recuperata tramite opportuna deallocazione

Reference Counting

- Nel nostro piccolo si può ipotizzare di utilizzare un "semplice" sistema di gestione della memoria, il *reference counting*...
- Basato sul conteggio "semiautomatico" del numero di puntatori che puntano ad una certa struttura dati
- Ogni volta che un nuovo puntatore punta alla struttura dati, ***incremento di reference count*** per quella struttura
- Ogni volta che un puntatore smette di puntare alla struttura dati, ***decremento di reference count*** per quella struttura
- In ogni momento si ha il controllo sul numero di puntatori che puntano alla struttura dati – se tale numero è maggiore di zero, la struttura non è puntata da nessuno e quindi è possibile deallocarla
- Per fare funzionare il tutto è necessaria MASSIMA DISCIPLINA...

Reference Counting

Come implementare un meccanismo di *reference counting*?

- Predisporre una struttura dati globale che rappresenti una tabella di due colonne e *n* (quante?) righe
- Nella prima colonna sono memorizzati ***gli indirizzi "puri" delle aree di memoria allocate***
- Nella seconda colonna sono memorizzati i ***reference count "rilasciati"***
- Per completare il tutto servono tre funzioni/procedure:
  - Una per **allocare** la memoria e impostare a **uno** il *reference count*
  - Una per copiare un puntatore in un altro → **incrementando** conteggio relativo a un indirizzo dato
  - Una per eliminare un "riferimento" → **decrementando** conteggio relativo a un indirizzo dato

Reference Counting

- Com'è implementata la tabella?
- Com'è implementata la ricerca nella tabella?
- ...e le "regole" di utilizzo?
- L'interfaccia:

```
void* referenceMalloc (size_t size);
void* copyReference(void* p);
void releaseReference(void *p);
```

…e l'implementazione?

UN PREMIO PER L'IMPLEMENTAZIONE MIGLIORE!

Gli esercizi precedenti

Come cambiano gli esercizi passati se si tiene conto della possibilità di allocare la memoria in modo dinamico?

- Merge Sort
- Calcolo determinante
- Rubrica

…facciamo le cose con ordine…

ADT – Abstract Data Type

- Specifica un insieme di dati e un insieme di operazioni applicabili a tali dati
- *Astratto* ➔ "funzionalmente" indipendente dalle varie possibili implementazioni
  - Le operazioni applicabili sono l'interfaccia dell'ADT
  - Un ADT può essere implementato in vari modi pur mantenendo la **stessa** interfaccia
- L'interfaccia fornisce un ***costruttore,*** che restituisce un ***handle*** astratto, e varie operazioni, rappresentate come funzioni che accettano come argomento tale *handle* astratto
- *handle* è di solito un puntatore a "qualcosa"

ADT – Abstract Data Type

- ADT è rappresentato da un'interfaccia, che nasconde l'implementazione corrispondente
- Gli utenti di un ADT utilizzano solo l'interfaccia ma non accedono (almeno non dovrebbero) direttamente all'implementazione poiché questa in futuro può cambiare
- Tutto ciò è basato sul principio di ***information hiding***, ovvero proteggere i "clienti" da decisioni di design che in futuro possono cambiare
- La potenza del concetto di ADT sta nel fatto che l'implementazione è **nascosta** al cliente: viene resa pubblica solo l'interfaccia
- Questo significa che un ADT può essere implementato in vari modi ma, finché non viene cambiata l'interfaccia, i programmi che lo utilizzano non devono essere alterati

Matrici come ADT

- Definire *handle* (tipo di base)
- Definire costruttore (e distruttore…)
- Definire operazioni

```
typedef double Element;
typedef struct DynMatrixStruct
{
    Element* matrix;
    int rows, cols;
} *DynMatrix;
```

DynMatrix *è un puntatore ad una struttura da allocare dinamicamente che contiene una matrice da allocare dinamicamente*

Costruttore

- Alloca la memoria necessaria, inizializza i dati in modo opportuno e restituisce un puntatore a *handle*

```
DynMatrix newDynMatrix(int rows, int cols)
{
    DynMatrix dm = (DynMatrix)
       malloc(sizeof(struct DynMatrixStruct));
    if (dm == NULL)
        return NULL;
    dm->matrix = (Element*)malloc(sizeof(Element)*rows*cols);
    if (dm->matrix == NULL)
        return NULL;
    dm->rows = rows;
    dm->cols = cols;
    return dm;
}
```

Distruttore

- Per deallocare un *DynMatrix* è sufficiente scrivere **`free(aDynMatrix)`**?
- Evidentemente no: occorre prima deallocare la matrice, poi la struttura contenitore → occorre una procedura che lo sappia fare

```
void destroyDynMatrix(DynMatrix m)
{
    free(m->matrix);
    free(m);
}
```


Operazioni

- Data una *DynMatrix*:
  - Verifica dei *bound* → data una coppia di valori (riga, colonna) verificarne la validità
  - Recupero numero di righe
  - Recupero numero di colonne
  - Recupero di un elemento
  - Assegnamento di un elemento
  - …altre funzioni non “primitive”

Operazioni – interfaccia

```
boolean checkBounds(DynMatrix m, int row,
                                    int col);

int getRowCount(DynMatrix m);

int getColCount(DynMatrix m);

Element getMatrixElement(DynMatrix m,
                         int row, int col);

void setMatrixElement(DynMatrix m, int row,
                      int col, Element value);
```


Verifica dei bound

```
boolean checkBounds(DynMatrix m,
                            int row, int col)
{
    return row >= 0 && col >= 0 &&
           row < m->rows && col < m->cols;
}
```

Dimensioni

```
int getRowCount(DynMatrix m)
{
    return m->rows;
}

int getColCount(DynMatrix m)
{
    return m->cols;
}
```


Funzioni accessorie

Consentono di leggere o impostare un elemento della matrice date riga e colonna

```
Element getMatrixElement(DynMatrix m, int row, int col)
{
    assert(checkBounds(m, row, col));
    return *(m->matrix + row * m->cols + col);
}

void setMatrixElement(DynMatrix m, int row, int col,
                                                Element value)
{
    assert(checkBounds(m, row, col));
    *(m->matrix + row * m->cols + col) = value;
}
```


Chi è assert?

- È una macro (per il preprocessore) che prende come argomento un valore booleano
- Tipicamente tale valore è il risultato del calcolo di una espressione
- Se il valore è vero, nessun problema, se è falso, stampa su **`stderr`** un messaggio d'errore (normalmente l'espressione) e termina (**`abort()`**) l'applicazione
- Può essere utile e comodo per verificare le *precondizioni* sull'invocazione di una funzione/procedura

Cosa sono le *precondizioni*?

- Le *precondizioni* sono ***condizioni sugli argomenti di una funzione/procedura*** che devono essere rispettate affinché l'invocazione di tale funzione/procedura vada a buon fine
- ***Dovrebbe essere il chiamante*** ad assicurarsi che le *precondizioni* (che fanno parte dell'interfaccia) siano rispettate
- A livello di chiamato si può (si dovrebbe) verificare che le *precondizioni* siano verificate e, se non lo sono, terminare "brutalmente" l'applicazione → non è possibile proseguire
- Servono
  - A garantire che l'applicazione non prosegua nell'esecuzione se non ci sono le condizioni affinché questa possa lavorare correttamente
  - Ad aiutare lo sviluppatore ad individuare eventuali bug

Stampa

```
void printMatrice_dyn(DynMatrix m)
{
    int row, col;
    char format[10] = "";
    strcat(format, ELEMENTFORMAT);
    strcat(format, "\t"); //Minimizzare il numero di spazi
    for (row = 0; row < getRowCount(m); row++)
    {
        for (col = 0; col < getColCount(m); col++)
            printf(format, getMatrixElement(m, row, col));
        printf("\n");
    }
    printf("\n");
}
```


Estrazione Minore

```
DynMatrix estraiMinore_dyn(DynMatrix m, int elRow, int elColumn)
{
    int row, col;
    DynMatrix minore;
    assert(getRowCount(m) == getColCount(m)); //matrice quadrata
    assert(checkBounds(m, elRow, elColumn));
    minore = newDynMatrix(getRowCount(m) - 1, getColCount(m) - 1);
    for (row = 0; row < getRowCount(m); row++)
    {
        int rowMinor  = (row < elRow) ? row : row - 1;
        for (col = 0; col < getColCount(m); col++)
        {
            if (row != elRow && col != elColumn)
            {
                int colMinor = col < elColumn ? col : col - 1;
                setMatrixElement(minore, rowMinor, colMinor,
                                 getMatrixElement(m, row, col));
            }
        }
    }
    return minore;
}
```


Determinante

```
Element determinante_dyn(DynMatrix m)
{
    int riga = 0, colonna;
    assert(getRowCount(m) == getColCount(m)); //matrice quadrata
    if (getRowCount(m) == 2)
    {
        return getMatrixElement(m, 0, 0) * getMatrixElement(m, 1, 1) -
            getMatrixElement(m, 0, 1) * getMatrixElement(m, 1, 0);
    }
    else
    {
        Element det = 0;
        for (colonna = 0; colonna < getColCount(m); colonna++)
        {
            DynMatrix minore;
            Element detMinore;
            minore = estraiMinore_dyn(m, riga, colonna);
            printMatrice_dyn(minore);
            detMinore = determinante_dyn(minore);
            det += getMatrixElement(m, riga, colonna) *
                        (colonna % 2 == 0 ? detMinore : -detMinore);
            destroyDynMatrix(minore);
        }
        return det;
    }
}
```


Domand**one**!

In tutte le funzioni di ADT di **`DynMatrix`** tranne che sulla **`newDynMatrix`** ***manca una precondizione su un argomento***; quale?

Rispost**one**!

- L’unico argomento non verificato è quello di tipo **`DynMatrix`**
- Se tale argomento è un puntatore non valido (**`==`** **`NULL`**), non si può lavorare
- …e se è un puntatore non valido e diverso da **`NULL`**?
- …purtroppo **non è possibile** verificare che un indirizzo assegnato ad un puntatore punti ad un’area di memoria valida