

Allocazione dinamica

- Quando non si sa a priori quanto un array debba essere grande → allocazione dinamica di array
- Quando si deve restituire un array come risultato di una funzione (a meno di non “barare” incapsulandolo in una struttura)
- Quando anche un array non basta perché una volta allocato ha dimensione fissa → allocazione dinamica di strutture ricorsive (che roba è!?)

1

Malloc

- La memoria viene allocata tramite una funzione:
`void* malloc(size_t dim);`
- Il valore di ritorno è un puntatore senza tipo (`void*`) ovvero un indirizzo all’inizio di un area di memoria la cui dimensione è definita solamente dall’argomento della `malloc` stessa
- Il sistema ricorda, per ogni puntatore allocato, 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!

2

Free

- La memoria viene deallocata tramite la procedura:
`void free(void* p);`
- Il sistema sa quanta memoria deallocare per quel puntatore
- 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

3

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 → la memoria viene allocata esplicitamente e deallocata automaticamente
- 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

4

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, viene incrementato il *reference count* per quella struttura
- Ogni volta che un puntatore smette di puntare alla struttura dati, viene decrementato il *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 quindi è possibile deallocarla
- Per fare funzionare il tutto è necessaria MASSIMA DISCIPLINA... Ma non più di quella che serve per evitare *memory leaking* a mano...⁵

Reference Counting

- Come implementare un meccanismo di *reference counting*?
 - La maniera più semplice è 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 due procedure
 - Una per **incrementare** il conteggio relativo ad un indirizzo dato
 - Una per **decrementare** il conteggio relativo ad un indirizzo dato

6

Reference Counting

- Com'è implementata la tabella?
- Com'è implementata la ricerca nella tabella?
- Quando incrementare il conteggio?
- Quando decrementare il conteggio?
- L'interfaccia:

```
void addReference(void* p);  
void releaseReference(void *p);
```

...e l'implementazione?

7

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
 - Menù
 - Rubrica

...facciamo le cose con ordine...

8

ADT – Abstract Data Type

- Specifica un insieme di dati ed un insieme di operazioni applicabili a tali dati
- *Astratto* nel senso che è “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 l'*handle* astratto
- L'*handle* è di solito un puntatore a “qualcosa”

9

ADT – Abstract Data Type

- Un ADT è rappresentato da un'interfaccia, che scherma l'implementazione corrispondente.
- Gli utenti di un ADT utilizzano solo l'interfaccia ma non accedono (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

10

Matrici come ADT

- Definire l'*handle* (il tipo di base)
- Definire il costruttore (e il distruttore...)
- Definire le 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

11

Costruttore

- Alloca la memoria necessaria, inizializza i dati in modo opportuno e restituisce un puntatore all'*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;  
}
```

12

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);
}
```

13

Operazioni

- Data una *DynMatrix*:
 - Verifica dei *bounds* → 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”

14

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);
```

15

Verifica dei bounds

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

16

Dimensioni

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

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

17

Accessors

- 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;
}
```

18

Chi è assert?

- È una macro (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
- È usato per verificare le *precondizioni* su una funzione/procedura

19

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

20

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");
}
```

*Indipendente
dall'implementazione!!!*

21

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;
}
```

*Indipendente
dall'implementazione!!!*

22

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;
    }
}
```

*Indipendente
dall'implementazione!!!*

23

Domandone!

- In tutte le funzioni viste prima tranne che sulla **newDynMatrix** manca una condizione; quale?

24

Rispostone!

- 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

25