Allocazione della memoria

In C++ è possibile creare (allocare) variabili in maniera **statica** o **dinamica**. Nell'allocazione **statica** una variabile esiste ed è utilizzabile dal momento della sua dichiarazione fino al termine del blocco nel quale è stata dichiarata (scope della variabile).

```
int Fatt(int n) {
    int fat=1;      // inizio esistenza variabile fat (allocazione)
    while (n>=1) {
        fat *= n;
        n--;
    }
    return fat;
}// fine esistenza fat (deallocazione)
```

Le operazioni di allocazione e deallocazione sono a carico del sistema.

Nell'allocazione **dinamica** una variabile viene allocata e deallocata esplicitamente con opportuni operatori.

Possiamo dunque dire che il C++ prevede due modi ben distinti per memorizzare i dati in memoria:

1. Attraverso l'uso di variabili: la memoria riservata per le variabili viene assegnata al compile time e non può essere modificata al runtime
2. Utilizzando l'allocazione dinamica della memoria: i dati vengono allocati secondo necessità nell'area di memoria denominata *heap* o *free store* (posta fra il codice del programma e lo stack).
   La memoria allocata dinamicamente viene determinata al runtime; quindi l'utilizzo della heap dà al programma la possibilità di creare spazio in RAM per i dati durante l'esecuzione (e a seconda delle circostanze…).
   NB: la heap ha estensione limitata: ogni tentativo di allocazione dinamica di memoria va ad erodere la disponibilità della heap, ed è possibile andare *out of memory*.

Il C++ mette a disposizione gli operatori **new** e **delete** per gestire l'allocazione e deallocazione dinamica della memoria. La sintassi è:

```
var_ptr = new tipo_var;
...
delete var_ptr;
```

L'operatore new alloca memoria sufficiente per contenere un valore del tipo tipo_var e restituisce un puntatore a tale valore. Se nella heap non c'è spazio sufficiente per questa allocazione, la new ritorna NULL (→ è sempre necessario verificare il valore di ritorno di una new).
L'operatore delete rilascia la memoria a cui punta var_ptr. Una volta rilasciata, la memoria può essere riallocata per scopi diversi.

Notare che un oggetto creato tramite new esiste finchè non viene esplicitamente distrutto dalla corrispondente delete. Questo porta a due considerazioni:

- Tramite new e delete è possibile gestire dati permanenti all'interno del programma (NB: diversi dai dati statici!)
- La memoria allocata con new non è disponibile per altre allocazioni fino a quando non viene liberata con delete → per evitare *memory leaks* è

necessario usare la delete ogni qualvolta un dato permanente non serve più. C++ non prevede nessun meccanismo di *garbage collection*, ossia di deallocazione automatica di memoria attraverso la ricerca di dati non referenziati. La garbage collection mette il programmatore al riparo da sprechi di memoria, ma ciò va a scapito delle performance.

Esempio

```
int f1()
{
    int *p;
    p = new int;    // allocazione di memoria per un integer
    if (!p) return ERROR;

    *p = 20;        // assegna a quell'area di memoria il valore 20
    cout << *p;

    delete p;       // rilascia l'area di memoria
    return 0;
}
```

Nota: il gestore della heap è uno strumento piuttosto primitivo, che normalmente non prevede la compattazione delle aree di memoria allocate. Allocazioni e deallocazioni causano la frammentazione della heap, situazione in cui il computer ha molta memoria libera ma nessun segmento sufficientemente grande per l'allocazione richiesta.

Allocazione di array

Le keywords new e delete hanno una sintassi diversa nel caso di allocazione e deallocazione di arrays.

```
var_ptr = new tipo_var[dimensione];
delete [] var_ptr;
```

Le parentesi quadre servono ad indicare al compilatore che deve essere deallocata la memoria per l'intero array.

⇒ Cosa succede eseguendo il seguente codice?

```
float *p;
p = new float[10];
...
delete p;
```

Allocazione dinamica della memoria - Compatibilità con il C

Anche il C gestisce l'allocazione dinamica della memoria.
Non ha gli operatori new e delete, ma utilizza le funzioni di libreria **malloc** e **free** (definite in stdlib.h) per allocare e deallocare la memoria dinamica.

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

dove num_byte è il numero di bytes che si desidera allocare, mentre size_t è un tipo definito che con la maggior parte dei compilatori corrisponde ad un unsigned int.
La malloc restituisce un void*, quindi è necessario un cast ad un puntatore al tipo di dato allocato. Esempio:

```
int *i;
i = (int*)malloc(sizeof(int));
```

La funzione malloc restituisce un puntatore alla prima regione di memoria libera allocata nella heap; se non esiste memoria sufficiente, ritorna un null pointer.

Notare l'utilizzo della funzione **sizeof** per ottenere la dimensione del tipo di dato integer.
La funzione free libera la memoria precedentemente allocata con una malloc:

```
void free(void *ptr);
```

L'esempio precedente, utilizzando malloc e free, diventa:

```
int f2()
{
    int *i;
    i = (int*)malloc(sizeof(int));
                    // allocazione di memoria per un integer
    if (!i) return ERROR;

    *i = 20;        // assegna a quell'area di memoria il valore 20
    cout << *i;

    free (i);       // rilascia l'area di memoria
    return 0;
}
```

Quindi malloc e free sono perfettamente in grado di gestire allocazione e deallocazione dinamica della memoria.
Il C++, pur mantenendo il supporto di queste funzioni per *backward compatibility*, definisce gli operatori new e delete per una serie di ragioni:

- new calcola automaticamente la dimensione del tipo di dato che viene allocato, quindi non è necessario utilizzare sizeof (→ non è possibile allocare una quantità errata di memoria dinamica)
- new restituisce automaticamente il tipo di puntatore corretto, quindi non è necessario un cast
- con new è possibile inizializzare il dato:

```
int *p;
p = new int(99);  // *p assume il valore 99
```

- tramite operator overloading, è possibile definire delle versioni personalizzate di new e delete.

NB: per evitare possibili incompatibilità, si sconsiglia di utilizzare new/delete e malloc/free all'interno dello stesso blocco di programma.

Esempio: la libreria CStash.

Si tratta di una utility per lo storage di dati; è simile ad un array, ma la dimensione è variabile, ossia può essere gestita al runtime.

La libreria è dotata di funzioni per:
- inizializzare
- aggiungere un elemento
- accedere ad un elemento
- contare gli elementi
- aumentare l'occupazione di memoria
- cancellare (liberare la memoria)

I dati vencono memorizzati utilizzando un `unsigned char*`, perché è il più piccolo elemento di memory storage messo a disposizione dal C++ (in molti compilatori corrisponde ad un byte).
La libreria è volutamente scritta *C-style* facendo uso di una struttura.

```
typedef struct CStashTag {
    int size;      // Dimensione di ogni elemento (in bytes)
    int quantity;  // Numero di elementi che può contenere
    int next;      // Prossimo elemento libero
    // Array di bytes allocato dinamicamente:
    unsigned char* storage;
} CStash;

const int increment = 100;
void initialize(CStash* s, int size);    // inizializza
int add(CStash* s, const void* element); // aggiunge un elemento
void* fetch(CStash* s, int index);       // accede ad un elemento
int count(CStash* s);                    // conta elementi
void inflate(CStash* s, int increase);   // aumenta dimensione
void cleanup(CStash* s);                 // libera la memoria
```

Inizializzazione:

```
void initialize(CStash* s, int size) {
    s->size = size;
    s->quantity = 0;
    s->storage = 0;    // Inizialmente non viene allocata memoria
    s->next = 0;
}
```

Aggiunta di un elemento:

```
int add(CStash* s, const void* element) {
    if(s->next >= s->quantity)
        inflate(s, increment);  // Se necessario amplia la struttura

    // Copia l'elemento nello storage (byte per byte),
    // iniziando alla prima posizione disponibile
    int startBytes = s->next * s->size;
    unsigned char* e = (unsigned char*)element;
    for (int i = 0; i < s->size; i++)
        s->storage[startBytes + i] = e[i];
    s->next++;
    return (s->next - 1); // Ritorna l'indice
}
```

La funzione aggiunge un elemento nella prima posizione disponibile. Per prima cosa verifica se c'è ancora spazio nell'allocazione corrente, e in caso contrario espande l'allocazione di memoria con una chiamata alla funzione inflate. Quindi si occupa della copia dell'elemento nello storage, copiandolo byte per byte (non può fare un semplice assegnamento perché tutto ciò che la funzione riceve è un void*).

Accesso ad un elemento:

```
void* fetch(CStash* s, int index) {
    // Controlla se l'indice è "in bounds":
    if(index < 0 or index >= s->next)
        return 0;  // Se non lo è ritorna 0
    // Ritorna un puntatore all'elemento desiderato:
    return &(s->storage[index * s->size]);
}
```

La funzione per prima cosa verifica che l'indice sia "in bounds", poi ritorna un puntatore void all'elemento richiesto.

Conta elementi:

```
int count(CStash* s) {
    return s->next;    // Numero di elementi in CStash
}
```

Estensione allocazione di memoria

```
void inflate(CStash* s, int increase) {
    int newQuantity = s->quantity + increase;
    int newBytes = newQuantity * s->size;
    int oldBytes = s->quantity * s->size;

    // allocazione nuovo array
    unsigned char* b = new unsigned char[newBytes];

    // Copia da vecchio a nuovo array
    for(int i = 0; i < oldBytes; i++)
        b[i] = s->storage[i];

    delete [](s->storage); // Dealloca vecchio array
    s->storage = b; // Storage punta alla nuova locazione
    s->quantity = newQuantity;
}
```

La funzione alloca un array di dimensione newBytes e copia in quella locazione il vecchio array byte per byte. Fatto questo dealloca l'area di memoria del vecchio array e setta il dato storage di CStash in modo che punti al nuovo array.
In questo modo la struttura può contenere increase elementi in più.

Deallocazione

```
void cleanup(CStash* s) {
    if(s->storage != 0)
        delete[] s->storage;
}
```

La funzione si limita a chiamare la delete[] che si occupa della deallocazione di memoria.

Utilizzo di CStash

```
#include "CLib.h"
using namespace std;

int main() {
    CStash intStash;
    int i;

    // Inizializzazione: specifichiamo che si tratta di una struttura che
    // contiene degli interi
    initialize(&intStash, sizeof(int));
    for(i = 0; i < 150; i++)
        add(&intStash, &i);    // ad ogni locazione assegniamo
                               // un numero progressivo

    for(i = 0; i < count(&intStash); i++)
        cout << "Elemento " << i << " = "
        << *(int*)fetch(&intStash, i)
        << endl;

    // fine: deallocazione memoria
    cleanup(&intStash);
    return 0;
}
```

Alcune considerazioni

L'implementazione di CStash, per quanto efficiente, è piuttosto contorta.
Tutte le funzioni della libreria accettano un puntatore a CStash, quindi ogni volta che facciamo uso della struttura dobbiamo passarne l'indirizzo.

I nomi delle funzioni delle libreria (initialize, add, count, …) sono del tutto generici, ma cosa succede se facciamo uso di due diverse librerie nelle quali si inizializzano, aggiungono e contano elementi? Come possiamo evitare il conflitto di nomi?
Nelle librerie C generalmente non si usano funzioni dai nomi così generici: nel nome delle funzioni si fa riferimento al tipo di dato.

```
void CStash_initialize(CStash* s, int size);
int  CStash_add(CStash* s, const void* element);
int  CStash_count(CStash* s);
…
```

Quindi in generale nelle librerie C il tipo di dato tende a comparire sia nel nome delle funzioni che nei loro parametri formali.