

## Programmazione 2 - Prof. Bolzoni

Il corso prosegue i contenuti di programmazione 1.

Si userà C come linguaggio di riferimento.

2 argomenti nuovi rispetto a proj. 1

- gestione della memoria dinamica
- programmazione in ricorsione

Gestione di memorie dinamiche permette di diventare più efficienti, di non sprecare risorse. Es. fino ad ora abbiamo visto la gestione delle memorie statiche: quando servivano delle variabili si dichiaravano tali variabili ad array, del quale si metteva a dichiarare anche la dimensione. Ad es. se si dichiarava un array di interi di 100 elementi si prevedeva che tale numero servisse soltanto alle esigenze del nostro programma.

Ma se non conosco a priori il numero dei valori che dovrò andare a gestire, la dimensione del mio array è solo una stima, un'ipotesi. Potrò avere sotto un'intuizione corretta e la dimensione del mio array essere quella giusta, ma potrei anche aver allocato troppe risorse rispetto alle esigenze (ad es. se fosse stato sufficiente un array di dimensioni 10) e in questo caso ho sprecato risorse; di contro potrebbe emergere nel corso dell'esecuzione del programma che ci sarebbe stata la necessità di avere un array di 1000 elementi, e in questo caso mi trovo impossibilitato a gestire le richieste del mio programma. Esempio: un ero di portare che guardavo il video: se alloco risorse per 100 persone e lo guardano solo in 10 ho sprecato risorse, ma se lo guardano in 1000 non ho abbastanza risorse per gestire un numero di utenti che non potevo conoscere a priori. Io voglio che il programma funzioni perché sono stato fortunato ad avere la giusta intuizione nel prevedere un dato, ma voglio che funzioni correttamente in ogni caso, e senza sprecare risorse.

Ad es., quando si scriveva la funzione per la stampa di un array, si scriveva in modo che funzionasse per un array di qualsiasi dimensione, per quanto tale dimensione non fosse conosciuta a priori. Questo lo si farà in generale per tutti i programmi, quindi non avremo nessuna precisazione sul numero dei dati che

suddiviso a gestire: potremo avere pochissimi dati come l'ultimissimi dati e il numero di questi dati non sarà in nessun modo cablato all'interno del codice.

L'ricorsione è una tecnica di programmazione che, al per sé, non ci permette di fare nulls di nuovo o nulla di più rispetto all'iterazione. Però molte volte risolvere problemi in ricorsione è molto più semplice che farlo in iterazione; molti problemi sono risolvibili con pochissimo codice in ricorsione, al contrario del codice che devo scrivere in iterazione; altre volte è molto più intuitiva la soluzione in ricorsione rispetto a quella in iterazione.

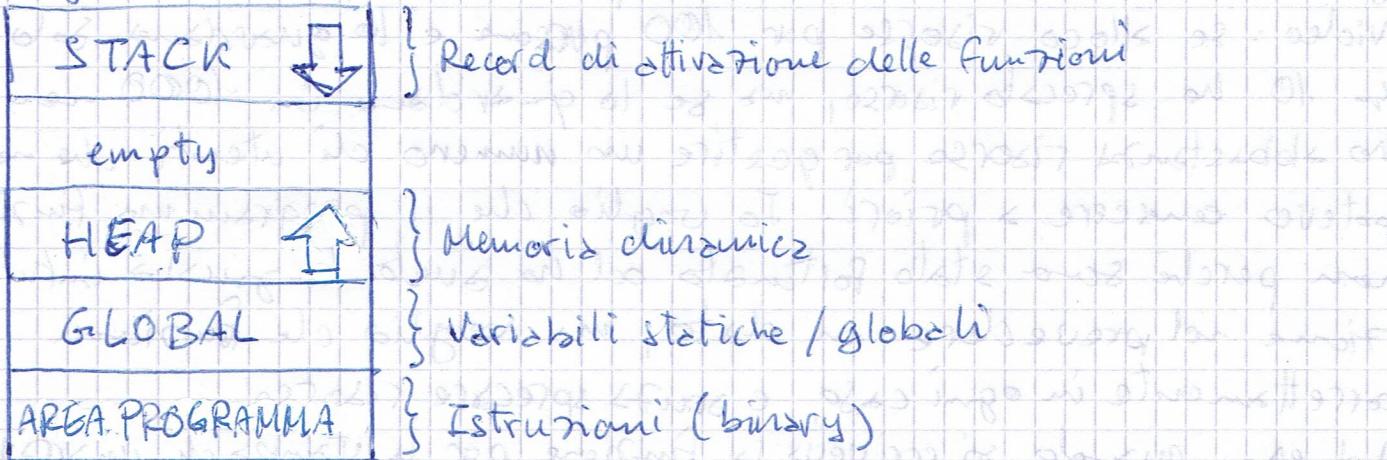
In programmazione 1 si sono visti i costrutti di base della programmazione:

- 1) Sequenza: un'operazione dopo l'altra
- 2) Scelta (o alternativa o condizione): il costrutto if ... else
- 3) Iterazione (cioè i cicli): while, do ... while e for

Con i cicli si fa la programmazione in iterazione, che è un modo di programmare complementare alla programmazione in ricorsione.

Vediamo ora come viene suddivisa la memoria durante l'esecuzione di un programma in C e di come la struttura della memoria si evolve in base alle istruzioni.

La struttura della memoria di un programma in C è la seguente:



- **Area programma**: contiene il codice binario, ovvero il codice generato dal compilatore quando si usa a compilare il codice sorgente.

- Area global: anche quest'area ha una dimensione che rimane immutata per tutta l'esecuzione del programma. L'area global contiene le variabili statiche e globali: quando si lancia l'eseguibile del programma la dimensione dell'area global può essere calcolata, in quanto il numero di variabili di questo tipo presenti nel programma è definito, come pure il loro tipo.

Le altre due aree, stack ed heap, sono delle zone di memoria dinamica, ovvero durante l'esecuzione di un programma possono aumentare o diminuire in dimensione a seconda di ciò che avviene nel programma.

- Stack: area deputata a raccogliere e gestire tutti i record di attivazione delle funzioni. Ogni volta che una funzione diventa attiva (e quindi è in esecuzione nel programma) necessita del suo record di attivazione per poter lavorare.
- Heap: è l'area deputata alla gestione della memoria dinamica, cioè di quelle aree che sono esplicitamente richieste o esplicitamente liberate.

Global e heap contengono variabili alle quali è possibile accedere da qualunque parte del programma, non solo e soltanto da una specifica funzione: ovvero queste due aree non contengono mai variabili locali. Al contrario, questo avviene per lo stack.

Heap e stack sono variabili non soltanto nella dimensione, ma anche nel contenuto.

Vediamo cosa c'è effettivamente nello stack, ovvero di cosa sia un record di attivazione.

In un record di attivazione altro non è che lo spazio di lavoro di una funzione, cioè contiene tutto ciò di cui una funzione necessita per lavorare; si tratta essenzialmente di tre cose:

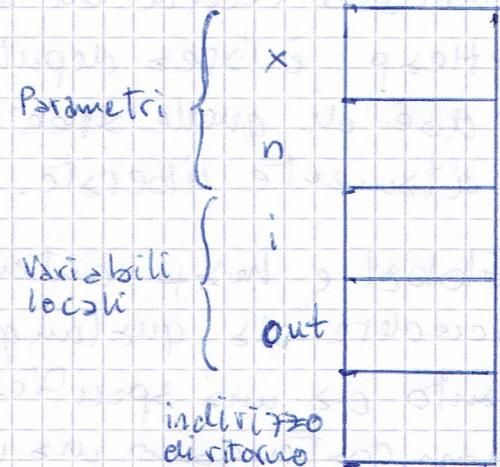
- Parametri
- Variabili locali
- Indirizzo di ritorno

Di questi tre elementi, uno solo è essenziale e sempre necessario, ovvero l'indirizzo di ritorno. Parametri e variabili locali possono esserci come non esserci: ad esempio, non c'è nessun obbligo che una funzione riceva dei parametri in input, e

analogamente anche le variabili locali non sono strettamente necessarie: ci saranno se la funzione le utilizza, altrimenti no. L'indirizzo di ritorno invece è sempre necessario: quando una funzione viene invocata, una volta che viene eseguito tutto il suo codice e si arriva alla terminazione della funzione (return o la gratta finale), il programma deve ritornare nel punto esatto in cui è stata invocata la funzione, quindi bisogna che si sappia dove ritornare, cioè deve essere stato memorizzato quale sia l'indirizzo del codice da cui bisogna proseguire. Consideriamo un esempio: una semplice funzione potenza che riceve in input due interi e restituisce il valore della potenza che si ottiene considerando il primo intero ricavato come base e il secondo intero come esponente.

```

4
5 int potenza (int x, int n)
6 {
7     int i;
8     int out = 1;
9     for (i=0; i < n; i++)
10    {
11        out = out * x;
12    }
13    return out;
14 }
```



Record di attivazione della funz. potenza

La funzione riceve due parametri ( $x$  ed  $n$ ), quindi necessita di due celle di memoria per memorizzare questi parametri; analogamente, la funzione utilizza due variabili locali ( $i$  ed  $out$ ), quindi necessita di due celle di memoria per queste due variabili; in più abbiamo una cella di memoria per l'indirizzo di ritorno. Ogni volta che la funzione potenza sarà invocata, si metterà sullo stack un record di attivazione con questa struttura, ovvero con queste cinque celle; queste celle verranno poi riempite a seconda del momento di esecuzione.

All'interno di queste celle non c'è una cella specificamente per il valore di ritorno. In questo caso, quando la funzione termina, deve ritornare il valore contenuto nella cella di  $out$  al punto del

programma che aveva invocato la funzione, ovvero al punto il cui indirizzo è contenuto nella cella "indirizzo di ritorno".

Ricordiamoci che il record di attivazione sussiste sol esistere quando una funzione viene invocata e sussiste di esistere quando la funzione termina (cioè viene eliminata dallo stack e lo spazio viene liberato).cioè quando la funzione termina, le celle che contenevano i valori usati dalla funzione perdono di significato e non sono più accessibili: quindi un ipotetico cella per il valore di ritorno sarebbe inutile, visto che verrebbe subito eliminata con la fine della funzione e dunque anche tale valore andrebbe perduto.

Come avviene quindi il ritorno di un valore? Il valore di ritorno viene restituito e reso accessibile tramite un registro macchina, che è una memoria separata dalla RAM, che continua sol esistere e sol essere accessibile in qualunque momento, indipendentemente dalla funzione che ne ha scritto il valore.

Quindi al termine della funzione il valore di ritorno viene scritto in questo registro macchina, l'esecuzione del programma riprende nel punto indicato dal valore contenuto nella cella dell'indirizzo di ritorno, il record di attivazione della funzione viene rimosso dallo stack e il programma legge il valore ritornato dal registro macchina, che continua sol essere accessibile anche dopo la rimozione del record di attivazione della funzione.

Il valore di ritorno non è sempre obbligatorio: sol es., funzioni di tipo VOID non restituiscono nessun valore di ritorno.

Per quella che è la semantica del C, se ci si dimentica di mettere il return al termine di una funzione che prevede un valore di ritorno, comunque il programma viene compilato senza produrre errori e comunque durante l'esecuzione, il programma ritorna qualcosa: il "qualsiasi" che viene ritornato sarà anche in questo caso il contenuto del registro macchina. Il comando return infatti va a mettere uno specifico valore nel registro macchina, ma comunque, anche indipendentemente dal comando return, il registro macchina avrà un contenuto. Oltre che per contenere il valore di ritorno infatti, il registro macchina viene utilizzato anche per le operazioni aritmetiche: quindi il registro macchina conterrà l'ultimo valore che vi è

è stato inserito per l'esecuzione di una qualche operazione algebrica. Quindi il valore di ritorno potrebbe essere quello desiderato, se tale valore per un caso fortuito fosse proprio l'ultimo valore che vi è stato inserito, anche se non venisse esplicitamente specificato il comando return. Però non vogliano che il programma dica il risultato corretto semplicemente per fortuna; quindi per accorgersi di aver dimenticato il return in una funzione che lo prevederebbe, visto che il compilatore generalmente eseguibile, è buona norma compilare usando il parametra -Wall, che visualizza, oltre che gli errori, anche i warning, cioè un avviso della presenza di situazioni potenzialmente problematiche, ma non bloccanti: sarà quindi per compito del programmatore indagare queste situazioni potenzialmente problematiche e decidere come gestirle.

Approfondimento: il registro memoria utilizzato per il valore di ritorno, che viene usato anche per le operazioni algebriche, contiene normalmente il risultato dell'ultima operazione algebrica eseguita. Consideriamo il seguente programma:

```
#include <stc16.h>
int potenza ( int base , int esponente )
{
    int i;
    int risultato = 1;
    for ( i=0; i < esponente; i++ )
        risultato = risultato * base;
    return risultato;
}
int main ()
{
    int base, esponente, risultato;
    printf ("Digita un valore per la base: ");
    scanf ("%d", &base);
    printf ("Digita un valore per l'esponente: ");
    scanf ("%d", &esponente);
    risultato = potenza ( base, esponente );
    printf ("% elevato a %d = %d \n", base, esponente, risultato);
```

```
    return 0;
```

```
}
```

Se lo si compila e lo si esegue, ipotizzando che l'utente inserisca 2 come valore per la base e 5 come valore per l'esponente si ottiene come risultato la scritta

$$2 \text{ elevato a } 5 = 32$$

Supponiamo di dimenticarsi il comando "return risultato;" alla fine della funzione: compilando ed eseguendo il programma, supponendo che l'utente digitò anche in questo caso 2 come valore per la base e 5 come valore per l'esponente, si ottiene

$$2 \text{ elevato a } 5 = 5$$

Perché? Per rispondere a questa domanda basta chiedersi quale sia l'ultima operazione algebrica che la funzione compie. Si potrebbe essere indotti a pensare che l'ultima operazione sia "risultato = risultato \* base;" e che quindi il valore contenuto nel registro macchina dovrebbe essere quello dell'elevamento a potenza, ma in realtà l'ultima operazione è l'incremento della variabile contatore, nel nostro caso da 2 a 5. Ecco spiegato il valore ritornato.

Se, per ipotesi, al posto di "return risultato;" scrivessimo un ciclo vuoto del tipo "for (i=0; i < risultato; i++); " il contenuto del registro macchina al termine della funzione sarebbe proprio il risultato dell'elevamento a potenza e il valore ritornato sarebbe comunque quello desiderato, anche in assenza di una return esplicita.

Definita la struttura di un record di attivazione, diciamo a parlare dello stack.

Lo stack non è altro che una zona di memoria dedicata a contenere i record di attivazione delle funzioni. Lo stack può venire modificato in due modi

- Aggiungendo un nuovo elemento in cima allo stack (push)
- Rimuovendo l'elemento in cima allo stack (pop)

L'aggiunta o l'eliminazione dei record di attivazione si fa sempre in cima allo stack: quindi, nel momento in cui faccio un'operazione

di push, aggiunge un record di attivazione in cima alla pil<sup>s</sup>, cioè allo stack; nel momento in cui si effettua un'operazione di pop, si va ad eliminare solo e soltanto il record di attivazione in cima alla pil<sup>s</sup> (modalità di gestione del tipo LIFO: last in, first out).

Quindi quando il programma comincia l'esecuzione, nello stack è presente il solo record di attivazione del main; quando il main invoca una funzione, nella cima della pil<sup>s</sup> viene aggiunto un nuovo record di attivazione (cioè sopra al record di attivazione del main); se questa funzione effettua una chiamata ad una nuova funzione, il record di attivazione di questa nuova funzione va sincrono in cima allo stack. L'unica funzione attiva in qualunque momento è quella che ha il record di attivazione in cima alla pil<sup>s</sup>: solo quella può terminare, dunque solo il record di attivazione in cima alla pil<sup>s</sup> può venire eliminato.

L'heap invece è quella zona di memoria dedicata per gestire le richieste esplicite di allocazione o di disallocazione dinamiche di memoria: ci sono due comandi specifici in C per allocare o deallocare zone di memoria in quest'area: malloc e free.

Quindi il programma potrà più fare richieste specifiche di allocazione di memoria tramite il comando malloc e specificando di quanto memoria ha bisogno; tramite il comando free invece è possibile sudare a liberare zone di memoria precedentemente allocata in maniera dinamica.

Contrariamente allo stack, la heap non può "crescere" in modo sequenziale: si può rimuovere uno qualunque degli elementi presenti (non solo il primo della pil<sup>s</sup>, come avviene per lo stack), mentre l'aggiunta di un nuovo elemento avviene in una zona degna a contenervolo. In sostanza, per quanto riguarda la heap, non si è in alcun modo vincolati a dover liberare la memoria in base all'ordine con cui la si è richiesta; quindi la heap può avere dei "buchi" al suo interno, cioè zone di memoria non più occupate comprese tra zone ancora occupate. Quindi una nuova richiesta può trovare posto in un "buco" creato in precedenza, oppure il buco potrebbe non essere abbastanza grosso, e quindi l'allocation dovrà avvenire in una zona diversa.

Facciamo un confronto tra memoria stack e memoria heap

### Stack:

- Accesso molto veloce
- Non è necessario disallocare esplicitamente la memoria
- Lo spazio è gestito in modo efficiente dalla CPU e la memoria non verrà frammentata
- Contiene solo variabili locali
- Limite sulla dimensione dello stack (dipendente dal sistema operativo)
- Le variabili non possono essere ridimensionate

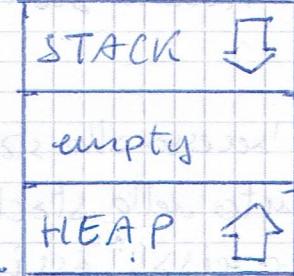
### Heap:

- Accesso relativamente più lento
- Il programmatore è incaricato di gestire la memoria (dove allocare e liberare le variabili)
- Nessun uso efficiente garantito dello spazio (la memoria può frammentarsi nel tempo)
- Le variabili sono accessibili a livello globale
- Nessun limite alla dimensione della memoria
- Le variabili possono essere rideimensionate usando realloc()

L'accesso allo stack è molto veloce perché, di tutto ciò che è contenuto nello stack il programma può accedere soltanto ad una parte ben precisa, ossia l'ultimo (cioè che è in cima); non c'è necessità di disallocare esplicitamente memoria, dato che quando la funzione termina, tutto il suo record di attivazione viene liberato immediatamente senza bisogno che lo faccia manualmente il programmatore; dato che non si eliminano blocchi intermedi della pila, ma sempre e solo il blocco (= record di attivazione) in cima alla pila, la memoria non avrà mai dei buchi, cioè non sarà mai frammentata; lo stack contiene solo e soltanto variabili locali, ovvero variabili accessibili esclusivamente alla funzione proprietaria di quell' specifico record di attivazione: tutte le altre var possono vedere queste variabili, né tantomeno lavorarci sepr.; lo stack ha una dimensione massima dipendente dal sistema operativo: questo comporta che il numero massimo teorico di chiavi di funzioni è un valore finito, dato che ogni chiamata di funzione alloca un record di attivazione nello stack: avendo lo stack una dimensione massima fissa dal sistema operativo, sarà dunque limitato anche il numero massimo di record di attivazione che è possibile allocargli, quindi il numero di chiamate è limitato; da

ultimo, le variabili, una volta messe sullo stack, non possono essere ridimensionate, quindi avranno una dimensione fissa.  
La heap di centro certe varie variabili accessibili a livello globale: quando vi si alloca uno spazio tramite una malloc questo spazio rimane allocato finché non viene esplicitamente liberato tramite una free oppure finché non termina il programma; dato che, per la caratteristica stessa della sua gestione, la heap può essere frammentata (cioè contenere dei buchi), l'accesso sia in lettura che in scrittura è più lento e richiede l'uso di indici per sapere dove ciascuna variabile si trova e dove ci sono zone di memoria libere; in fine esiste un comando in C, realloc, che permette di modificare la quantità di memoria assegnata a una variabile, aggiungendone di più o liberandone una parte.

Lo stack e la heap sono, in qualche modo, collegati tra di loro: tra di loro c'è una zona "vuota" che il sistema operativo dà al programma C per le sue esigenze, perché sa che potenzialmente il programma può richiedere dell' memoria che può aumentare, diminuire, aumentare di nuovo... > secondo di quale sarà l'esecuzione. Non necessariamente come queste richieste variano si può sapere a priori quando si esegue il programma, dato che l'esecuzione può essere guidata dai dati e i dati possono essere forniti tramite un input dall'utente o letti da file, e questi input non sono prevedibili. Quindi si "spreca" un certo quantitativo di RAM che potrebbe essere utile al programma (tanto di RAM sostanzialmente ce n'è sempre in abbondanza), ma si evita il fatto di doverne assegnare successivamente e di non avere memoria libera nelle zone di memoria contigua a quelli che ha assegnato per il programma, il che salva dalla necessità di dover riallocare memoria se ne presenta la necessità.



Vediamo ora un programma un po' più esteso che sfrutta la funzione potenza scritta precedentemente

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #int risultato; // variabile globale
4
5 int potenza (int x, int n)
6 {
7     int i = 0;
8     int out = 1;
9     for (i; i < n; i++)
10    {
11        out = out * x;
12    }
13    return out;
14 }
15
16
17 int main ()
18 {
19     int i = 4;
20     int out = 2;
21     risultato = potenza (i, out);
22     printf ("il risultato e': %d\n", risultato);
23
24     return 0;
25 }
```

main	i	4
	out	2
	x	4
	n	2
	i	0 → 1 → 2
	out	1 → 4 → 16
	Inizializzo o ritorno	a
	Heap	
	risultato	16
	code	

Quando il programma viene lanciato (il codice risiede nell'area di memoria deputata) viene allocato lo spazio per la variabile globale "risultato" e nello stack viene allocato il record di attivazione per il main, con le sue due variabili locali "i" ed "out". Attenzione: all'inizio del programma, le celle di memoria allocate per la variabile globale "risultato" non contiene stessa valore, dato che la variabile viene dichiarata, ma non viene fatto nessun assegnamento; invece le due variabili locali del main "i" ed "out" vengono dichiarate e contemporaneamente viene assegnato loro un valore: quindi nel record di attivazione del main vengono riservate due celle di memoria per queste due variabili, che subito vengono riempite con i valori 4 (la cella per "i") e 2 (la cella per "out").

Subito dopo il main fa un assegnamento alla variabile globale "risultato" invocando la funzione "potenza": chiamando la funzione potenza si alloca un record di attivazione per tale funzione, che andrà sopra il record del main (il verso di rieccupero dello stack è invertito: consideriamo "sopra" come se stessino guardando lo stack dall'alto verso il basso); quindi verranno allocate due celle di memoria per i parametri "x" ed "n" che la funzione riceverà in input, con i loro valori x=4 e n=2; poi due celle per le variabili locali "i" ed "out", anche esse assegnate contemporaneamente alla loro dichiarazione: i=0 e out=1; infine una cella

per l'indirizzo di ritorno (che nello schema si è indicato così). I parametri "x" ed "n" della funzione potenza ricevono il loro valore dalle variabili rispettivamente "i" ed "out" del main; sia nel main che nella funzione potenza abbiano delle variabili locali chiamate "i" ed "out", ma si tratta di variabili differenti: pur avendo lo stesso nome, la variabile "i" del main è una variabile diversa dalla "i" della funzione potenza, e così pure per le variabili "out". La funzione potenza, quando riunimata, dopo le due dichiarazioni ed assegnamenti delle variabili locali, farà una prima esecuzione del ciclo for: il valore di "out" passerà da 1 a 4, il valore di i da 0 a 1; dopo questo primo ciclo, la condizione di ciclo "i < n" sarà ancora valida, quindi il ciclo verrà eseguito una seconda volta, e il valore di "out" passerà da 4 a 16, mentre il valore di "i" da 1 a 2.

A questo punto il valore di "i" sarà uguale al valore di "n", quindi la condizione di ciclo non sarà più verificata e la funzione uscirà dal ciclo. Si passa quindi all'istruzione successiva, che è una return: il valore da ritornare viene scritto nel registro meccanico, l'esecuzione del programma ritorna all'indirizzo di contenuto nella cella dell'indirizzo di ritorno e il record di attivazione della funzione potenza viene "distrutto" (o meglio, la memoria viene deallocata); a questo punto il main fa l'eseguimento alla variabile globale "risultato" andando a prendere dal registro meccanico il valore di ritorno che vi era stato scritto dalla funzione "potenza" prima della sua terminazione.

A questo punto viene eseguito la printf, che è anch'essa una funzione di libreria: quindi anche per la printf ci sarà l'allocazione di un record di attivazione sullo stack, l'esecuzione della funzione, la liberazione della memoria precedentemente allocata per la printf sullo stack e il ritorno al main.

Nel main a questo punto si incontra la return: quindi il programma termina e tutta la zona di memoria che il sistema operativo aveva dato al programma per il suo funzionamento viene rilasciata, ed è possibile nuovamente gestirla da parte del sistema operativo.