Gestione della memoria a run-time

Codice macchina e dati entrambi in RAM, ma in zone separate:

- memoria per il codice macchina fissata a tempo di compilazione
- memoria per i dati locali alle funzioni (variabili e parametri) cresce e descresce dinamicamente durante l'esecuzione: viene gestita a **pila**

Una **pila** (o **stack**) è una struttura dati con accesso LIFO: Last In First Out = ultimo entrato è il primo a uscire (Es.: pila di piatti).

A run-time viene gestita automaticamente la **pila dei record di attivazione** (RDA) in memoria centrale:

- per **ogni attivazione di funzione** viene creato un nuovo RDA in cima alla pila
- al termine dell'attivazione della funzione il RDA viene rimosso dalla pila

Ogni RDA contiene:

- le locazioni di memoria per i parametri formali (se presenti)
- le locazioni di memoria per le variabili locali (se presenti)
- l'indirizzo di ritorno = indirizzo della prossima operazione da eseguire nella funzione chiamante

Esempio: file `funzioni/stack.c`

codice sorgente $\overset{\text{compilazione}}{\Longrightarrow}$ codice macchina (caricato in RAM al momento dell'esecuzione)

Supponiamo (per semplicità) che ad ogni istruzione del codice sorgente corrisponda una singola istruzione in linguaggio macchina:

	main()
	...
0A00	m1
0A01	m2
0A02	m3
0A03	m4
0A04	m5
0A05	m6
0A06	return
0A07	...

⇒ A(s)

	A()
	...
0B00	a1
0B01	a2
0B02	a3
0B03	a4
0B04	a5
0B05	a6
0B06	return
0B07	...

⇒ B(loc)

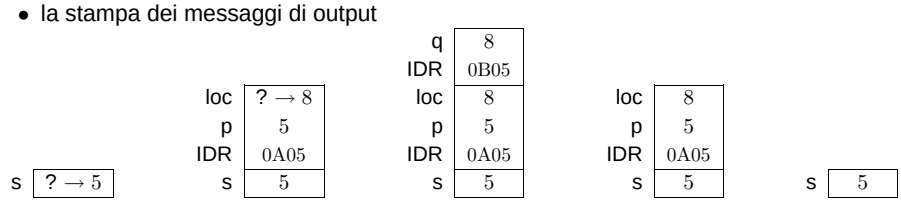
	B()
	...
0C00	b1
0C01	return
0C02	...

Due concetti fondamentali che determinano l'esecuzione:

- **program counter** (PC), che indica la prossima istruzione da eseguire
- **pila dei RDA**, con un RDA per ogni attivazione di funzione

Per seguire l'esecuzione del programma vediamo come evolvono in parallelo:

- la pila dei RDA (con indirizzo di ritorno)
- il program counter (in particolare durante le chiamate di funzione e i relativi ritorni)
- la stampa dei messaggi di output




```
Sono main()
Inserisci un intero: 5
Ora chiamo A(), con parametro attuale pari a 5
Sono A(). Il mio parametro p vale 5
Inserisci un intero: 8
Ora chiamo B(), con parametro attuale pari a 8
Sono B(). Il mio parametro q vale 8
Sono di nuovo A()
Sono di nuovo main()
```

Variabili automatiche e statiche

Tempo di vita di una variabile =
- = periodo in cui esiste la cella di memoria associata alla variabile
- = periodo in cui esiste il RDA corrispondente all'attivazione (per variabili **locali automatiche**)

Una variabile può anche essere **statica** $\Longrightarrow$ esiste per tutto il tempo di esecuzione del programma:
- se è dichiarata all'esterno di qualsiasi funzione, oppure
- se è locale ad una funzione e lo specificatore `static` precede la dichiarazione

  Es.: `void f(void) { static int x; ... }`
  - la variabile viene inizializzata alla prima attivazione della funzione
  - conserva il suo valore tra attivazioni successive
  - è locale, quindi visibile solo all'interno della funzione in cui è dichiarata

Esempio: Funzione che ritorna il numero di volte che è stata attivata.

Implementazione: file `funzioni/contatt.c`

Ricorsione

Una funzione che contiene al suo interno un'attivazione di sè stessa è detta **ricorsiva**.

Esempio: Programma che usa una funzione ricorsiva:

file `ricorsio/ricorsio.c`

Vediamo l'evoluzione della pila dei RDA per input 2. Output prodotto:

```
Sono main()
Inserisci un intero non negativo: 2
 - Attivo ricorsiva(2)
Sono ricorsiva(2) - Attivo ricorsiva(1)
Sono ricorsiva(1) - Attivo ricorsiva(0)
Sono ricorsiva(0) - Ho finito
Sono di nuovo ricorsiva(1) - Ho finito
Sono di nuovo ricorsiva(2) - Ho finito
Sono di nuovo main() - Ho finito
```

Cosa succede se si attiva `ricorsiva(-1)`?

Funzioni ricorsive sono convenienti per implementare funzioni matematiche che ammettono una **definizione induttiva**.

Esempio: fattoriale
- definizione iterativa: $fatt(n) = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$
- definizione induttiva:

$$fatt(n) = \begin{cases} 1, & \text{se } n = 0 \quad \text{(caso base)} \\ n \cdot fatt(n-1), & \text{se } n > 0 \quad \text{(caso ricorsivo)} \end{cases}$$

  È essenziale che applicando ripetutamente il caso ricorsivo, ci riconduciamo prima o poi al caso base.

algoritmo ricorsivo per il calcolo del fattoriale di un intero nonnegativo *n*
  if $n = 0$
  then return 1
  else calcola il fattoriale di $n - 1$
    moltiplicalo per *n*
    restituisci il valore ottenuto

Implementazione: file `ricorsio/fattoria.c`

Esercizio: Implementare le operazioni di somma, prodotto, ed esponente, utilizzando le seguenti definizioni induttive di tali operazioni.

- definizione induttiva di somma tra due interi nonnegativi:

$$somma(x,y) = \begin{cases} x, & \text{se } y = 0 \\ 1 + (somma(x, y-1)), & \text{se } y > 0 \end{cases}$$

- definizione induttiva di prodotto tra due interi nonnegativi:

$$prodotto(x,y) = \begin{cases} 0, & \text{se } y = 0 \\ somma(x, prodotto(x, y-1)), & \text{se } y > 0 \end{cases}$$

- definizione induttiva di elevamento a potenza tra due interi nonnegativi:

$$esponente(x,y) = \begin{cases} 1, & \text{se } y = 0 \\ prodotto(x, esponente(x, y-1)), & \text{se } y > 0 \end{cases}$$

Si possono inserire le definizioni delle funzioni nel file `ricorsio/driveint.c`.

Soluzione: file `ricorsio/operindu.c`

Esempio: Leggere una sequenza di caratteri terminata da '`\n`' e stamparla invertita.

Es.: `paolo\n` $\Longrightarrow$ `oloap`

Problema: prima di poter iniziare a stampare dobbiamo aver letto e memorizzato tutta la sequenza:

1. usando una struttura dati opportuna (array o lista) — più avanti
2. usando le celle di memoria della pila dei RDA come memoria temporanea

Implementazione: file `ricorsio/invertic.c`

Vediamo l'evoluzione della pila dei RDA con input "`abc\n`".

Esercizio: Leggere un intero e stamparne le cifre invertite (fornire una soluzione iterativa ed una ricorsiva).

Es.: `25138` $\Longrightarrow$ `83152`

Suggerimento: `25138 % 10` $=$ `8`
`25138 / 10` $=$ `2153`

Soluzione: file `ricorsio/invertin.c`

Esercizio: Leggere una sequenza di caratteri con un punto centrale, e verificare se è palindroma (ignorando gli spazi bianchi).

Una sequenza si dice **palindroma** se letta da sinistra a destra è identica a quando viene letta da destra a sinistra.

Es.: `i topi non av.vano nipoti`

Caratterizzazione induttiva di una sequenza palindroma:

- la sequenza costituita solo da '`.`' è palindroma
- una sequenza xsy è palindroma se lo è s e se x $=$ y.

Soluzione: file `ricorsio/palinric.c`

Cosa possiamo fare se la frase non contiene il '`.`' centrale?

Ricorsione multipla

Si ha ricorsione multipla quando un'attivazione di una funzione può causare **più di una attivazione ricorsiva** della stessa funzione.

Esempio: Funzione ricorsiva per il calcolo dell'n-esimo numero di Fibonacci.

Fibonacci: matematico pisano del 1200, interessato alla crescita di popolazioni.

Ideò un modello matematico per stimare il numero di individui ad ogni generazione:

$F(n)$ ... numero di individui alla generazione n-esima

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(n+2) &= F(n) + F(n+1) \end{aligned}$$

$F(0)$, $F(1)$, $F(2)$, ... è detta sequenza dei numeri di Fibonacci:
$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ \ldots$

Implementazione: file `ricorsio/fibonacc.c`

Esercizio: Aggiungere il codice per contare il numero di attivazioni ricorsive di `fibonacci`.

Esercizio: Fornire un'implementazione iterativa per il calcolo dell'n-esimo numero di Fibonacci.

Esercizio: Implementare in C la funzione di Ackermann $A(m,n)$ definita come segue:

$$A(m,n) = \begin{cases} n+1, & \text{se } m = 0 \quad \text{(caso base)} \\ A(m-1,1), & \text{se } n = 0 \quad \text{(caso ricorsivo)} \\ A(m-1, A(m,n-1)), & \text{altrimenti} \quad \text{(caso ricorsivo)} \end{cases}$$

Attenzione: cresce **molto** rapidamente (non elementare): $A(x,x)$ cresce piu' rapidamente di qualsiasi catena di esponenziali $2^{2^{\cdots^{2^x}}}$.

Soluzione: file `ricorsio/ackerman.c`

Esercizio: Implementare funzioni ricorsive sfruttando le seguenti definizioni induttive:

- massimo comun divisore

$$mcd(x,y) = \begin{cases} x, & \text{se } y = 0 \\ mcd(y,r), & \text{se } y > 0 \text{ e } x = q \times y + r, \text{ con } 0 \le r < y \end{cases}$$

  Soluzione: file `ricorsio/mcdricor.c`

- verifica se due numeri interi positivi sono primi tra loro

$$primi(x,y) = \begin{cases} vero, & \text{se } x = 1 \text{ oppure } y = 1 \quad \text{(caso base)} \\ falso, & \text{se } x \neq 1,\ y \neq 1 \text{ e } x = y \quad \text{(caso base)} \\ primi(x, y-x), & \text{se } x \neq 1,\ y \neq 1 \text{ e } x < y \quad \text{(caso ricorsivo)} \\ primi(x-y, y), & \text{se } x \neq 1,\ y \neq 1 \text{ e } x > y \quad \text{(caso ricorsivo)} \end{cases}$$

  Soluzione: file `ricorsio/primrico.c`

- resto della divisione tra un intero ed un intero positivo

$$resto(x,y) = \begin{cases} resto(x+y, y), & \text{se } x < 0 \quad \text{(caso ricorsivo)} \\ x & \text{se } 0 \le x < y \quad \text{(caso base)} \\ resto(x-y, y) & \text{se } x > y \quad \text{(caso ricorsivo)} \end{cases}$$

  Soluzione: file `ricorsio/restoric.c`

Esempio: Torri di Hanoi (leggenda Vietnamita).

- pila di dischi di dimensione decrescente su un perno 1
- vogliamo spostarla su un perno 2, usando un perno di appoggio 3
- condizioni:
  - possiamo spostare un solo disco alla volta
  - un disco più grande non può mai stare su un disco più piccolo
- secondo la leggenda: monaci stanno spostando 64 dischi; quando avranno finito, ci sarà la fine del mondo

Programma che stampa la sequenza di spostamenti da fare:

"muovi un disco dal perno x al perno y"

Idea: per spostare $n > 1$ dischi da 1 a 2, usando 3 come appoggio:

1. sposta $n - 1$ dischi da 1 a 3
2. sposta l'n-esimo disco da 1 a 2
3. sposta $n - 1$ dischi da 3 a 1

Implementazione: file `ricorsio/hanoi.c` (è un altro esempio di ricorsione multipla)

Visualizziamo l'albero delle attivazioni per 3 dischi.

Attenzione: quando si usa la ricorsione multipla, il numero di attivazioni ricorsive può essere **esponenziale** nella profondità delle chiamate ricorsive (cioè nell'altezza massima della pila dei RDA).

Esempio: Torri di Hanoi

$$\begin{aligned} att(n) &= \text{numero di attivazioni di } \texttt{muoviUnDisco} \text{ per } n \text{ dischi} \\ &= \text{numero di spostamenti di un disco} \end{aligned}$$

$$att(n) = \begin{cases} 1, & \text{se } n = 1 \\ 1 + 2 \cdot att(n-1), & \text{se } n > 1 \end{cases}$$

Senza 1 nel caso di $n > 1$ avremmo $att(n) = 2^{n-1}$.
$\Longrightarrow\ att(n) > 2^{n-1}$

È una caratteristica del problema (ovvero non esiste una soluzione migliore).

Esercizio: Contare il numero di attivazioni di `muoviUnDisco`.