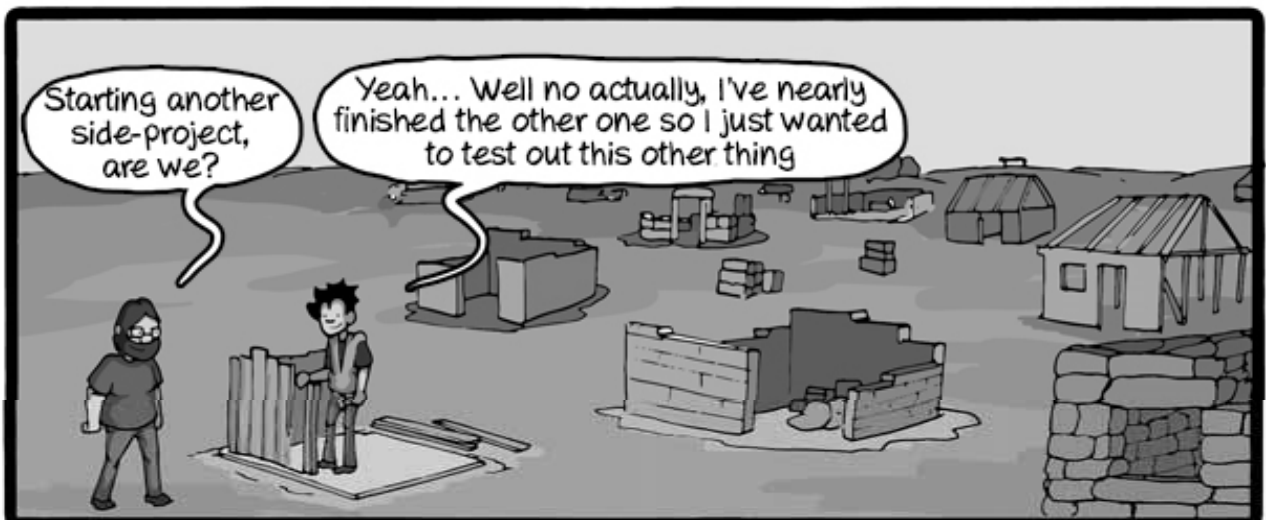


L 'ASTRAZIONE PROCEDURALE: i sottoprogrammi



Un sottoprogramma isola una porzione di codice
(composizione modulare del programma - dividi e conquista)



Sviluppo, manutenzione e riuso del codice

Sottoprogrammi funzionali e procedurali

Schema base di un programma.

program ::= directive_part

{ global_declarative_part }_{opt}

{ function_definition }₀₊

Costanti

Tipi,
variabili

*function_definition ::= type identifier ({ formal_parameters }_{opt})
{ local_declarative_part executable_part }*

local_declarative_part ::= type_declarations ?? / variable_declarations

int main () { *local_declarative_part* *executable_part* }

Tipi?
variabili

Sottoprogramma funzionale (funzione $y=f(x)$)

- $f(x)$ va definita prima della sua invocazione -> compilazione
- riceve dei valori (attraverso i parametri formali) e restituisce un valore;
- è invocata nei contesti dove è possibile una *expression*:
$$y=f(x) \quad \text{if } ((f(x)+5) \leq 10)$$
- contiene sempre un blocco $\{.....\}$ con:
 - dichiarazioni identificatori locali (regole di definizione dei globali);
 - istruzioni
 - l'istruzione **return** (*expression*)
- non può definire un'altra funzione innestata (vedi *function declarative part*)
- main è una funzione speciale;

Esempio

```
int sum(int x, int y)
{return (x+y);}
```

```
int main()
{ int a=10; int b=11;
  ...
  if (sum(a,b) > 20) printf ("high"); else printf("low");
  ... printf("\n%d ", sum (a+1,b));
}
```

I parametri formali definiscono l'interfaccia della funzione

tiporis nomefunction(*[*tipo1 nome1, tipon nomen*]**opt*) {.....}

I parametri attuali definiscono i dati che vengono passati

function_call ::= nomefunction (*{*actual_parameters*}**opt*)

1. numero parametri formali uguale al numero di quelli attuali in ogni esecuzione;
 2. corrispondenza posizionale tra parametri formali e attuali;
 3. il nome di un parametro formale può NON corrispondere a quello del corrispondente parametro attuale;
 4. tipo dei parametri formali: tipo base, identificatore di tipo (per ora NON la sua definizione) e può essere una struct, un puntatore (NO array);
 5. modalità di passaggio parametri è PER VALORE: ogni coppia parametro attuale/formale rispetta le regole di compatibilità e di conversione definite per gli assegnamenti;
 6. parametro attuale può essere una costante, variabile o espressione
 7. sottoprogramma restituisce un valore al sottoprogramma chiamante di tipo base, struct, puntatore, void (NO array).
- Ordine di valutazione dei parametri è implementation dependent
Es. printf(“...%d ... %d”, n++, n*4)
Effetti collaterali?

Parametri e sincronizzazione

```
int sum(int x, int y) {return (x+y);}
```

```
int main()  
{ int a=10; int b=11;  
  if (sum(a,b) > 20) printf ("high"); else printf("low");  
  ... printf("\n%d ", sum (a+1,b));  
}
```

	area dati globali vuota	
Macchina main		Macchina Sum
main (dati locali) a 10 b 11		sum (dati locali) Risultato ? x ? y ? ret_adr ?
codice main 100: if (sum(a,b) > 20) 101 printf(...);		Codice sum { return (x+y);}

PC =100

Che succede quando incontra **printf("\n%d ", sum (a+1,b)**

I sottoprogrammi procedurali

Un sottoprogramma che non restituisce un valore di ritorno al programma chiamante.

- $f(x)$ va definita prima della sua invocazione

Testata: **void** $f(\text{int } x)\{\dots\}$

Conseguenze:

- “return” non ha il valore da restituire;
 - “return valore” in generale genera warning del compilatore \Rightarrow valore eliminato
 - “return” è **opzionale** (la funzione comunque termina quando incontra la “}”)
- la funzione invocata come un’istruzione: $f(a);$

Es.

```
int sum(int x, int y) {.....return (x+y);}
```

```
void StampaSomma (int x, int y) {....printf(“%d”, sum(x,y));}
```

```
int main()
{  int a=10, b=11;
    ...
    StampaSomma(a,b);
}
```

Cosa cambia tra le due tipologie di uso del concetto di funzione

Se una funzione deve ritornare più valori?

Unica alternativa è di permettere ai parametri tra () di:

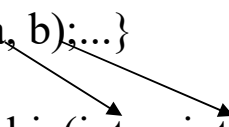
- passare un valore alla funzione chiamata
- MA ANCHE DI restituire “un valore” alla funzione chiamante

Ma come?

Esempio

```
int main ()
{int a=10; int b=11;
  scambia(a, b);...}

void scambia(int x, int y)
{int temp;
  temp=x; x=y; y=temp;
}
```



Passaggio parametri per indirizzo “simulato”

- parametro formale: si aspetta un indirizzo: tipo puntatore (*x)
- parametro attuale passa un indirizzo: tipo indirizzo (&)
- il parametro deve essere utilizzato all’interno delle funzioni tramite la dereferenziazione (es. *x=10) – accesso via indirizzo;

```
int main ()
{int a=10; int b=11;
  scambia(&a, &b);...}

void scambia(int *x, int *y)
{int temp;
  temp=*x; *x=*y; *y=temp;
}
```

Osservazioni:

- per ogni parametro formale si deve decidere se si aspetta un valore o un indirizzo
- obbligatoria per gli array;
- conveniente per risparmiare memoria con grandi strutture dati.

Passaggio vettori a funzioni

- Non si può passare un vettore, ma l'indirizzo di un elemento del vettore (in genere il primo)
- Formulazioni sintattiche alternative nei parametri e nelle istruzioni

Approccio semplice (ricordiamo di indentare il codice)

Non così

```
void reverse(char s[]) {int c,i,j;for(i=0,j=strlen(s)-1;i<j;i++,j--){c=s[i]; s[i]=s[j]; s[j]=c;}
```

Ma

```
void reverse (char s[])  
{ int temp, i, j;  
  for (i = 0, j = strlen(s) - 1; i < j; i++, j--)  
    {temp= s[i]; s[i] = s[j]; s[j] = temp; }  
}
```

```
int main()  
{ char vet[12]; // vettore con al più 11 caratteri utili  
  ....  
  reverse (vet);  
}
```

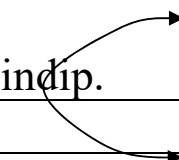
Alternative

- Parametro formale: char s[] oppure char *s;
- Parametro attuale: nome del vettore- sinonimo di indirizzo primo elemento (vet)
- Riferimento nella funzione ad un elemento del vettore: s[i] o *s

In generale?

```
typedef int Tarray[8];  
Tarray A;
```

Parametri formali	int f(int *a) int f(Tarray a) equivalenti int f(int a[])
Referenza all'interno di f	*a oppure a[i] equivalenti

indip. 

A proposito di stile

```
void scambia(int *x, int *y)....
```

```
void reverse (char s[])
```

```
{ int temp, i, j;  
  for (i = 0, j = strlen(s) - 1; i < j; i++, j--)  
    {temp= s[i]; s[i] = s[j]; s[j] = temp; }
```



```
void reverse (char s[])
```

```
{ int i, j;  
  for (i = 0, j = strlen(s) - 1; i < j; i++, j--) scambia(&s[i], &s[j]);  
}
```

Altro esempio

```
int main(){
```

... carica vettore

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
	0	1	2	3	4	5	6	7

```
void S1(int *a)
```

```
{ int I;
```

```
  *a=33; // a[0]=33;    ⇐ accesso all'ancora
```

```
  a[3]=22;              ⇐ spostamento relativo di 3 elementi
```

```
  printf("\n");
```

```
  for (I=0;I<=7;I++) printf(" %d",a[I]); scansione a partire dall'ancora
```

```
}
```

Invocazione

S1(A); ≡ S1(&A[0])

	a=A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
	0→33	1	2	3→22	4	5	6	7

↑ stampa

⇒|

```
void S2(int a[])
```

```
{ int I; *(a+2)=44; // a[2] ⇐ spostamento relativo di 2 elementi
```

```
  printf("\n"); for (I=0;I<=7;I++) printf(" %d",*(a+I));
```

```
}
```

Invocazione

S2(A);

	a=A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
	33	1	2→44	22	4	5	6	7

↑ stampa

⇒|

Invocazione

S2(&A[2]);

	A[0]	A[1]	a=A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
	33	1	44	22	4→44	5	6	7

↑ stampa

⇒|

Necessità di precisare con vettori:

- Indirizzo elemento iniziale
- Numero di elementi da considerare

Esempio su vettore caricato con “n” elementi.

```
double v[50];
```

```
double mul(double a[ ], int n) //moltiplica gli elementi di un array
{ int i; double ris= 1.0;
  for ( i = 0; i < n; i++ ) ris = ris * a[i];
  return ris;
}
```

Invocazione del main

mul(v, 50)	v[0]*v[1]* ... *v[49]
mul(&v[5], 7)	v[5]*v[6]* ... *v[11]
mul(v+5, 7)	v[5]*v[6]* ... *v[11]
mul(v,70)	

Matrici come parametri

Suggerito

...

```
typedef int riga[nc];  
riga mat[nr];
```

P.formale

```
void f (riga *M oppure riga M[] oppure int M[][nc])  
{... for (i=0; i<nr; i++)  
    for (j=0; j<nc; j++)  
        printf (“ %d”, M[i][j]);  
}
```

Invocazione

```
int main ()  
{ .... f(mat); oppure f(&mat[0]);}
```

Non suggerito

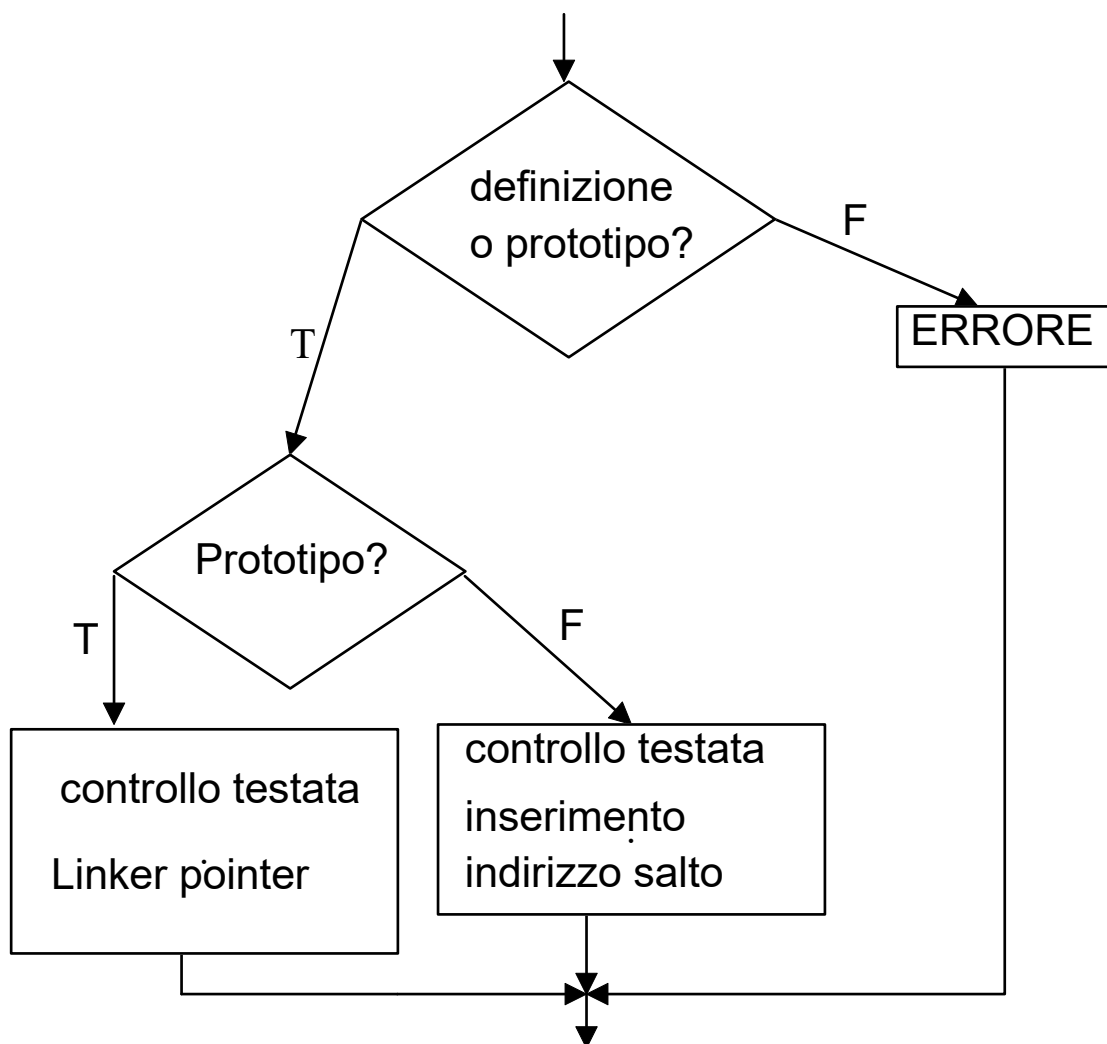
```
f(...) {for (i=0; i < (nr*nc); i++) printf (“ %d”,*((*M)+i);)}
```

Per invocare una funzione prima della definizione
(prototipo/dichiarazione di una funzione)

```
double mul(double a[ ], int n);
```

```
void main () { ... Z = mul(v,50); ... }
```

Regole della compilazione.



Effetti collaterali nell'uso delle variabili globali

Funzioni senza effetti

```
#define low 5
int a=2,b=1, ris;.....
int sum(int aa, int bb){ return(a+b); }
...
if (sum(a,b)>=low) printf ("sum=%d is over low", sum(a,b));
else printf("sum= %d is under low", sum(a,b));
```

Funzioni con effetti indesiderati

```
while (getchar()!='\n')
{c=getchar();....
}
```

Funzioni con effetti desiderati

```
int main()
{ int progressivo=1;
  int insert(int val)
  //insert in file;
  progressivo++;
}
```

La funzione main ha parametri?

Standard form

```
int main (int argc, char *argv[])
{int i;
  for (i=0;i<argc;i++) printf("%s\n",argv[i]);
}
```

Invocazione

```
>>./a.out                                nessun parametro
    argc = 1
    ./a.out
```

```
>>./a.out uno due                        tre parametri
    argc = 3
    ./a.out
    uno
    due
```

- Dopo ultima stringa NULL
- Interprete comandi

Come avviene l'invocazione?

Regole di associazione tra nomi e oggetti



Regole di visibilità (regole di scope) dei nomi (identificatori)

Uno stesso nome può essere associato a oggetti diversi in regioni diverse di uno stesso programma.

Es.

```
int i;  
int main() { int i; i= 3;}
```

Come si individua l'oggetto interessato quando viene invocato un nome in un'istruzione di una regione del programma?

Approcci:

- Scope statico: le regole dipendono dalla sola struttura sintattica del programma – verificabile a compile time
- Scope dinamico: le regole dipendono dal flusso di esecuzione a run-time.

Regole di scope statico

Blocco

Block ::= {block_declarative_part executable_part}

block_declarative_part ::= type_declarations / variable_declarations

- Ogni funzione contiene almeno un blocco
- un blocco può definire altri blocchi all'interno (annidati, paralleli)

Esempio:

```
#include <stdio.h>
```

```
typedef struct {int c1; float c2;} T;
```

```
void F1(int x);
```

```
int main()
```

```
{ T a;
```

```
    {int b; ... /*blocco 1*/ }
```

```
    {char c; ... /*blocco2*/ }
```

```
    ...
```

```
}
```

```
void F1(int x)
```

```
{T d;
```

```
    {int e;    /*blocco 3*/
```

```
        {const int f=4;    /*blocco4 */
```

```
        }
```

```
    }
```

```
    x=3; /*istruzione 1*/
```

```
    T=4; /*istruzione 2*/
```

```
    J=5; /*istruzione 3*/
```

```
    F1(4); /*istruzione 4*/
```

```
}
```

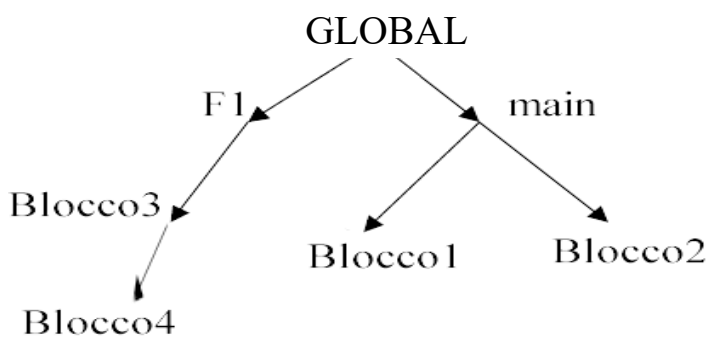
Definizioni:

- nomi globali \equiv nomi definiti in *global_declarative_part* incluso nomi di funzioni;
- nomi locali di un blocco \equiv nomi definiti nel blocco
- nomi locali di una funzione \equiv nomi definiti nella *function_declarative_part* + parametri formali;

Regole di base

- il blocco/funzione coincide con il campo di validità dei suoi nomi locali;
- se un blocco/funzione contiene altri blocchi, il campo di validità dei nomi del blocco/funzione più esterno si estende ai blocchi innestati;
- il campo di validità dei nomi globali coincide con l'intero programma.

Struttura gerarchica del campo di validità



unità	nomi definiti	campo validità
global	T, F1, main	il programma
main	a	main, blocco1, blocco2
blocco1	b	blocco1
blocco2	c	blocco2
F1	x, d	F1, blocco3, blocco4
blocco3	e	blocco3, blocco4
blocco4	f	blocco4

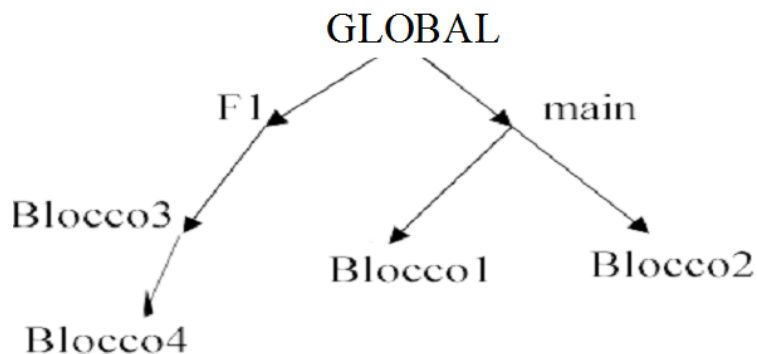
Regola campo di validità non è sufficiente

```
int i;                                doppia i
int main() { int i; i= 3;}
```

Navigazione gerarchia per l'associazione nome - oggetto

Dato un nome N in un'istruzione I di un blocco/funzione S:

- si cerca definizione di N tra quelle locali al blocco/funzione S
- se non esiste in S si cerca definizione nel blocco/funzione nel quale è stato dichiarato il blocco S; la navigazione può proseguire sino al blocco più esterno;
- se non la si trova si cerca tra i nomi globali;
- la ricerca termina quando:
 - si trova la prima definizione per il nome
 - la definizione per il nome non viene trovata nella navigazione - “undefined symbol error” in compilazione.



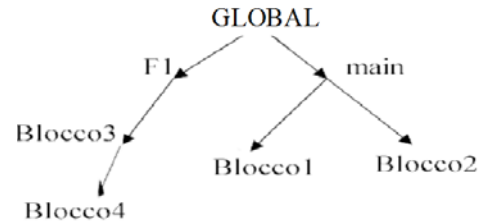
Avvertenza: una volta trovata l'associazione va verificata la congruenza semantica tra definizione e uso.

Esempio:

```
#include <stdio.h>
```

```
typedef struct {int c1; float c2;} T;
```

```
void F1(int x);
```



```
int main()
```

```
{ T a;
```

```
    {int b; ... /*blocco 1*/ }
```

```
    {char c; ... /*blocco2*/ }
```

```
    ...
```

```
}
```

```
void F1(int x)
```

```
{T d;
```

```
    {int e; /*blocco 3*/
```

```
        {const int f=4; /*blocco4 */
```

```
        }
```

```
    }
```

```
    x=3; /*istruzione 1*/
```

```
    T=4; /*istruzione 2*/
```

```
    J=5; /*istruzione 3*/
```

```
    F1(4); /*istruzione 4*/
```

```
}
```

Esempio

Le istruzioni della funzione F1 sono corrette?

- 1.
- 2.
- 3.
- 4.

Allocazione e tempo di vita delle variabili (RDA e stack)

Variabili globali

- allocate staticamente dal compilatore a inizio esecuzione programma;
- tempo di vita \equiv tempo di esecuzione del programma;

Variabili locali di un blocco/funzione

Linguaggi senza ricorsione

- Approccio statico: compilatore alloca tutte le variabili locali
- Approccio dinamico: allocazione delle variabili locali quando il blocco entra in esecuzione (tempo di vita \equiv tempo di esecuzione del blocco/funzione)

Più veloce il primo, ma il secondo usa meno memoria

Linguaggi con ricorsione (funzione si autoinvoca)

Impossibile l'approccio statico

L'approccio dinamico

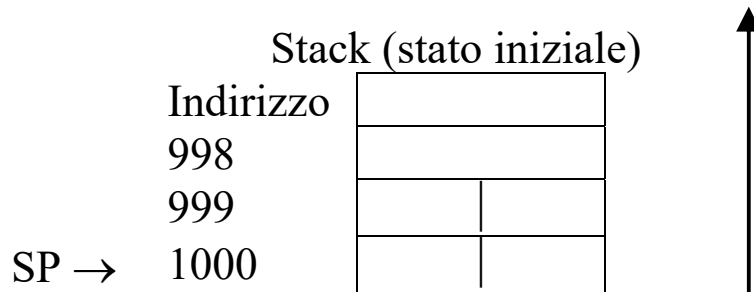
Il record di attivazione (RDA)

Area dati dedicata al singolo sottoprogramma/blocco

- valore di ritorno una funzione (return);
- parametri della funzione;
- registro per referenziare variabili locali
- indirizzo di ritorno;
- variabili locali;
- link statico per lo scope (non trattato oltre)

L'area Stack

RDA allocato/deallocato nello stack in modo automatico



Lo stack può essere vuoto o avere più RDA (se una funzione attiva un'altra funzione al proprio interno)

Estendiamo il linguaggio assembly

Istruzioni:

[W:] ADD oper1,oper2	somma in oper2
[W:] MOV oper1, oper2	copia in oper2
[W:] BR I	salta all'istruzione in parola con etichetta I
[W:] BREQ oper, I:	se oper =0 allora SALTA a istruzione I
[W:] JTS I	eseguire funzione che inizia all'etichetta I
[W:] RTS	ritorno da funzione
[W:] EXIT	termina esecuzione programma
[W:] READ oper:	carica valore letto da tastiera in oper
[W:] WRITE oper:	scrive su video valore in oper

Modalità di indirizzamento operandi

Un operando di add, mov, breq, read, write può essere specificato come

- #X valore del simbolo X
- X contenuto della parola di memoria con etichetta X
- Ri contenuto del registro Ri ($R7 \equiv SP$)
- (Ri) contenuto della parola di memoria il cui indirizzo è nel registro Ri

Direttive (pseudoistruzioni)

[X:] RES N Alloca N parole consecutive in memoria e associa l'indirizzo simbolico X(etichetta) alla prima parola

END [X] fine programma con etichetta della prima istruzione da eseguire

Esecuzione di una funzione

```
    int A, B, C;
int sum(int p1, int p2)
3.  {int temp;
4.   temp = p1+ p2;
5.   return(temp); }
void main ()
1.  {A=2; B=3;
2.   C = sum(A,B);
```

A: .RES 1 allocazione statica delle 3 variabili globali

B: .RES 1

C: .RES 1

STACK: .RES 1000 allocazione dello stack

//IN etichetta la prima istruzione eseguibile del main

IN: MOV #STACK, SP

ADD #999, SP inizializzazione SP

MOV #2, A 1)

MOV #3, B 1)

//invocazione della funzione sum

ADD #-1, SP 2) spazio RDA per il risultato

MOV A, (SP) ADD #-1,SP 2) spazio RDA per parametri

MOV B, (SP) ADD #-1,SP 2)

JTS SUM 2) invocazione sum

Stato dello stack SP->

all'atto della

Invocazione

di sum

valore di B
valore di A
risultato

// quando la funzione ha eseguito RTS

RET: ADD #2, SP 2) elimina parametri da RDA

ADD #1, SP 2) risultato ritornato in C e

MOV (SP), C 2) elimina spazio risultato

EXIT

Cosa accade quando il main esegue l'istruzione JTS:

- caricamento dell'indirizzo di ritorno nello stack e modifica del PC

MOV #RET, (SP)

ADD #-1, SP

MOV #SUM, PC

- prologo della funzione

sum: MOV R0, (SP)

ADD #-1, SP

salva registro R0

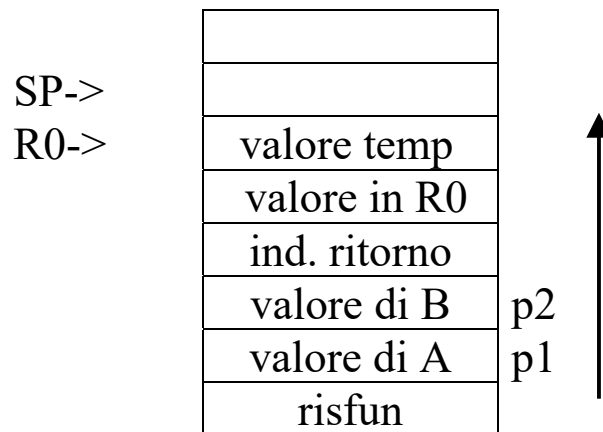
MOV SP, R0

carica valore di SP in R0

ADD #-1, SP

3) spazio RDA per variabile locale

// RDA ha raggiunto massima estensione



Esecuzione istruzioni della funzione

// temp=p1+p2 scomposta in temp=p1; temp=temp+p2;

MOV R0, R1

4) R1 = indirizzo di riferimento

ADD#4, R1

4) R1 contiene l'indirizzo assoluto di p1

MOV (R1), (R0)

4) temp=p1

MOV R0, R1

4) R1 = indirizzo di riferimento

ADD #3, R1

4) R1 contiene l'indirizzo assoluto di p2

ADD (R1), (R0)

4) temp=temp+p2

//return temp sposta risultato

MOV R0, R1	5) R1 = indirizzo di riferimento
ADD#5, R1	5) R1 contiene l'indirizzo assoluto di risfun
MOV (R0), (R1)	5) risfun = temp

//return temp predispone il ritorno riportando lo stack allo stato iniziale

ADD #1, SP	5) elimina var locale da RDA
ADD #1, SP	5) ripristina R0 togliendolo da RDA
MOV (SP), R0	5)
RTS	5)

Cosa accade quando la funzione esegue RTS

- viene prelevato dall'RDA l'indirizzo di ritorno e inizializzato il PC

ADD#1, SP
MOV (SP), PC

Programma condensato

A: .RES 1

B: .RES 1

C: .RES 1

STACK: .RES 1000

IN: MOV #STACK, SP

ADD #999, SP

MOV #2, A

MOV #3, B

ADD #-1, SP

MOV A, (SP)

ADD #-1, SP

MOV B, (SP)

ADD #-1, SP

JTS SUM

RET: ADD #2, SP

ADD #1, SP

MOV (SP), C

EXIT

sum: MOV R0, (SP)

ADD #-1, SP

MOV SP, R0

ADD #-1, SP

MOV R0, R1

ADD#4, R1

MOV (R1), (R0)

MOV R0, R1

ADD #3, R1

ADD (R1), (R0)

MOV R0, R1

ADD#5, R1

MOV (R0), (R1)

ADD #1, SP

ADD #1, SP

MOV (SP), R0

RTS

.END IN

Accenni alla programmazione ricorsiva

Calcolo $N!$ $N=0$ $N!=1$

$N>0$ $N!=N*(N-1)!$

↓

$(N-1)*(N-2)!$

↓

$(N-2)*(N-3)!$

↓

.....

*/*Soluzione ricorsiva diretta**

```
#include <stdio.h>
```

```
int N,R;
```

```
int fatt(int n)
```

```
{if (n == 0)
```

```
    return(1);
```

```
    else return (n * fatt(n-1));}
```

```
void main()
```

```
{printf("\nvalore di n: "); scanf("%d", &N);
```

```
    R= fatt(N);
```

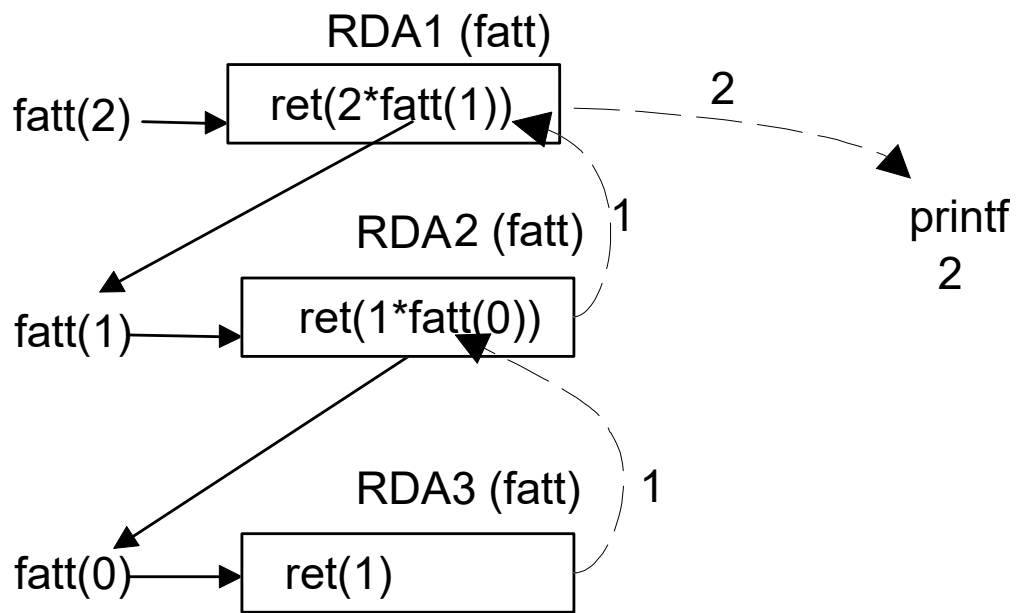
```
    printf("il fattoriale di %d è %d", N, R);
```

```
}
```

Formulazione ricorsiva di algoritmi:

- identificare uno o più sottocasi che definiscono la terminazione;
- determinare il passo ricorsivo: sottocaso del problema tale per cui la soluzione del sottocaso \equiv alla soluzione del problema, ma su un insieme ridotto di dati.

Esecuzione ricorsiva per N=2



Esempio 2: serie di Fibonacci (modello di crescita)

$$F = \{f_0, \dots, f_n\},$$

$$f_0 = 0$$

(caso base)

$$f_1 = 1$$

(caso base)

$$\text{Per } n > 1, f_n = f_{n-1} + f_{n-2}$$

(passo risorsivo)

da cui per esempio

$$f_0 = 0$$

$$f_1 = 1$$

$$f_2 = f_1 + f_0 = 1 + 0 = 1$$

$$f_3 = f_2 + f_1 = 1 + 1 = 2$$

$$f_4 = f_3 + f_2 = 2 + 1 = 3$$

$$f_5 = f_4 + f_3 = 3 + 2 = 5$$

Calcolo numero di Fibonacci di indice n

```
int fibonacci(int n)
```

```
{ if (n == 0) return 0;
```

```
  else if (n == 1) return 1;
```

```
    else return fibonacci(n-1) + fibonacci(n-2);
```

```
}
```

Algoritmo iterativo

```
int fibonacci(int n)
```

```
{ int ultima,penultima, corrente, i;
```

```
  if (n==0) return 0;
```

```
  if (n==1) return 1;
```

```
  ultima=1; penultima=0;
```

```
  for(i=2; i<=n; i++)
```

```
    {corrente=ultima+penultima;
```

```
      penultima=ultima; ultima=corrente;
```

```
    }
```

```
  return (corrente);
```

```
}
```

Esempio 3. /* Legge sequenza di 100 numeri e la visualizza in ordine inverso senza usare vettore*/

```
#include <stdio.h>
```

```
int i = 1, max=100;
```

```
void sequenza ()
```

```
{ int numero; scanf ("%d",&numero);
```

```
  if (i==max)
```

```
    {printf("-%d",numero); return;}
```

```
  else
```

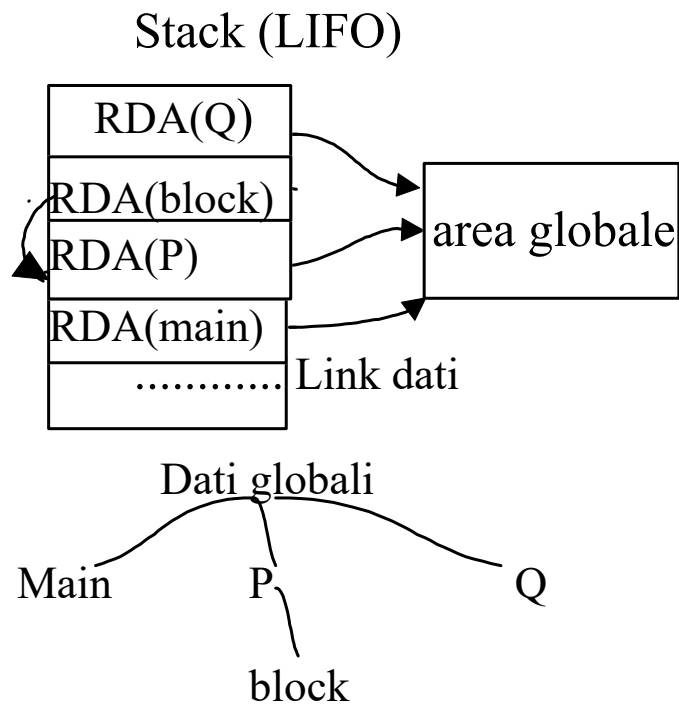
```
    {i++; sequenza(); printf("-%d",numero);}
```

```
}
```

```
main() { sequenza();}
```

Stack e RDA con blocchi e ricorsione

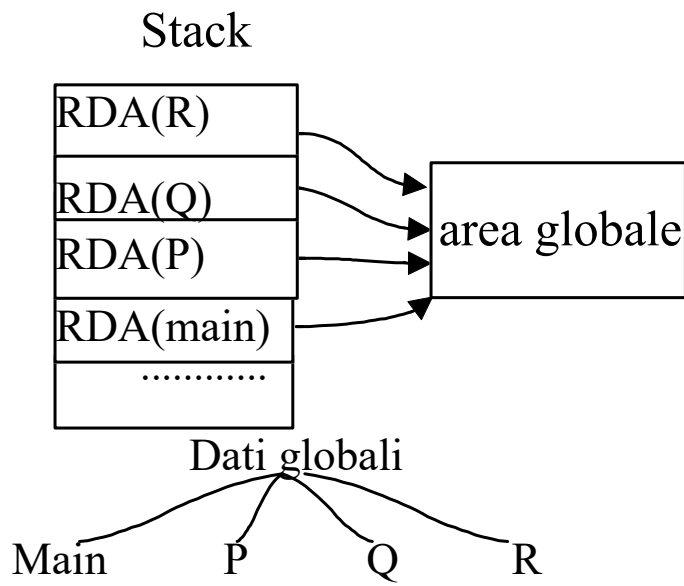
```
int main()
{...P; }
void P()
{...
  /*block*/
  Q;
}
}
void Q()
{...}
```



```

void main()
{...P; }
void P
{...Q; }
void Q()
{...R; }
void R()
{....}

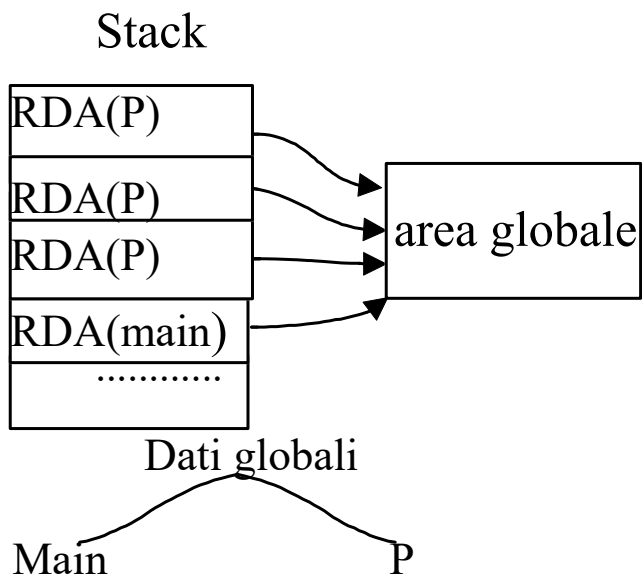
```



```

void main()
{...P; }
void P
{...P; }

```



Indicare, rispettando il formato di stampa previsto che cosa apparirà al video.

```
#include <stdio.h>
```

```
void F(int Y, int LIM)
```

```
{ if (Y == LIM) return;
```

```
  if (Y<LIM) {printf("\nY=%d lim= %d", Y, LIM); return;}
```

```
  printf("\nY= %d",Y); F(Y-2,LIM); printf("\n%d", Y);
```

```
}
```

```
int main(){F(12,6); printf("\nfine attivazione"); }
```

Stampa:

```
Y=10Y=8810fine attivazione
```