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* $\}_{0+}$

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

function_definition::=type identifier **(**$\{$ *formal_parameters* $\}_{opt}$**)**
{*function_declarative_part executable_part***}**

function_declarative_part ::= constant_declarations | *type_declarations* | *variable_declarations*

local_declarative_part ::= constant_declarations | *type_declarations* | *variable_declarations*

Sottoprogramma funzionale (funzione)

- va definita prima della sua invocazione

- riceve dei valori (attraverso i parametri formali) e restituisce un valore;

- è invocata nei contesti dove è possibile una *expression*:

- 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 *local_declarative_part)*

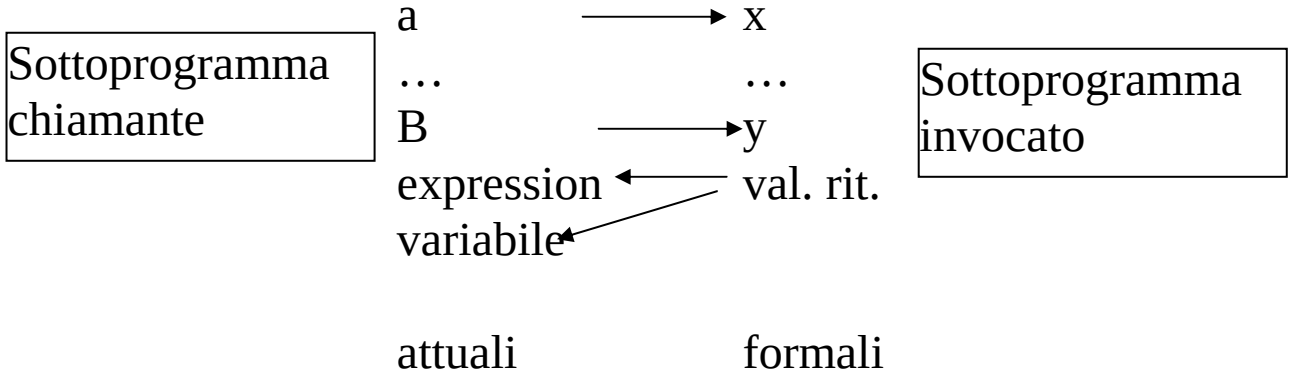
- main è una funzione speciale;

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,b));
}
```

I parametri



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* }$_{opt}$)

- numero parametri formali illimitato e uguale al numero di quelli attuali in ogni attivazione;
- corrispondenza posizionale tra parametri formali e attuali;
- tipo dei parametri formali: identificatore del tipo e NON la sua definizione e può essere un tipo base, una struct, un puntatore (NO array);
- ogni coppia parametro attuale/formale rispetta le regole di compatibilità e di conversione definite per gli assegnamenti;
- modalità di passaggio parametri: PER VALORE:
  - il parametro rappresenta un canale monodirezionale verso il sottoprogramma;
  - il valore di ritorno è l'unico dato che può essere restituito dalla funzione;
  - il parametro attuale può essere una variabile, costante o espressione;
- tipo del risultato: tipo base, struct, puntatore, void (NO array);
- il nome di un parametro formale può NON corrispondere a quello del corrispondente parametro attuale.

- Ordine di valutazione dei parametri è implementation dependent

  Es. printf("…%d … %d", n++, n*4)

Esecuzione esempio con sum (il record di attivazione):

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

PC?

I sottoprogrammi procedurali

Un sottoprogramma che non restituisce valori espliciti al programma chiamante (anche se può modificare in modo indiretto le variabili – vedi poi).

- utilizzo del tipo void per il risultato della funzione (es. void main()..)
- return opzionale (la procedura termina quando incontra la "}" che chiude il blocco della funzione
- la funzione invocata come un'istruzione
- l'invocazione procedurale di un sottoprogramma funzionale implica la perdita del valore di ritorno (es., scanf()).

Attenzione all'ordine di definizione)

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

void  StampaSomma (int x) {printf("……%d", x);}

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

Passaggio parametri per indirizzo “simulato”

I parametri per restituire più valori alla funzione chiamante

Esempio
Richiesta numero iscritti e aula di un esame

Appello(codices ↓ aula ↑ iscritti ↑)

1) valori ritornati in un record:

```
typedef struct   {int aula; int iscritti;} result_ty;

result_ty Appello(int codice)
  {result_ty temp;
   select DB (codice) ⇒ temp.aula, temp.iscritti
   return temp;
   }
```

2) Simulazione passaggio parametri per indirizzo

- parametro formale: tipo puntatore (*P) al tipo di struttura ricevuta;
- parametro attuale di tipo indirizzo (&) alla struttura dati passata;
- il parametro deve essere utilizzato nelle istruzioni tramite la dereferenziazione (es. *P=10) – accesso via indirizzo;

Osservazioni:
- obbligatoria per gli array;
- conveniente per risparmiare memoria con grandi strutture dati.

Esempio

```
#include <stdio.h>
int iscritti, aula, code;

void Appello(int c, int *a, int *i)
  {select DB (c) ⇒ *a, *i;}

int main()
{…..
   Appello(code, &aula, &iscritti);
printf("\ncode=%d, iscritti=%d, aula=%d", code, iscritti,aula);
}
```

Vettori e funzioni

- Parametro deve indicare l'indirizzo di un elemento (ancora)
- Formulazioni sintattiche alternative nei parametri e nelle istruzioni

Esempio 1

```
void itoa(int n, char *s)    s – inizio vettore
{int i, sign;
  if ((sign = n) < 0)    n = -n;   /* rende n positivo, sign x segno */
  i = 0;
  do {/* genera le cifre nell'ordine inverso */
         s[i] = n % 10 + '0'; i++;                    /* estrae la cifra seguente */
       } while ((n /= 10) > 0);   /* elimina cifra da n */
   if (sign < 0))    s[i] = '-';
       i++;     s[i] = '\0';
       reverse(s);
}
```

```
void reverse(char *s)   // convenzione stringhe
{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; }
}
```

Se intero max 32 bit ⇒ [-2147483648 a 2147483647] ⇒ 11 caratteri
⇒ vettore di 12 caratteri con convenzione \0

```
int main()    //prova
{int c=-123456789; char vet[12];
 itoa(c,vet);
 printf("%s",vet);
}
```

Esempio 2:

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

void S1(int *a) ≡ void S1(Tarray a)   ≡ 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 di 7 elementi a partire dall'ancora
 }

void S2(int a[])
 { int I; *(a+2)=44;              ⇐ spostamento relativo di 2 elementi
    printf("\n"); for (I=0;I<=7;I++) printf(" %d",*(a+I));
 }

void main()
{int I; printf("\n\n ");       for (I=0;I<=7;I++)A[I] =I;
```

S1(A); ≡ S1(&A[0])

esecuzione

	a=A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
S1	0->33	1	2	3→22	4	5	6	7

⇑ stampa ⇒|

S2(A);

esecuzione

	a=A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
S2	33	1	2→44	22	4	5	6	7

⇑ stampa ⇒|

S2(&A[2]);

esecuzione

	A[0]	A[1]	a=A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
S2	33	1	44	22	4→44	5	6	7

↑stampa ⇒|

}

Come contenere il problema?

Passare indirizzo elemento iniziale e il numero di elementi da considerare

Esempio:

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

…

typedef elemento riga[nc];

riga mat[nr];

Definizione

void F(riga m[], int nc, int nr,…);

o ?

void F(riga *m, int nc, int nr, …)

o ?

void F(elemento m[][nc], int nc, int nr, …);

o ?

void F(elemento *m, int nc, int nr, …);

Invocazione

F(mat); … F(&mat[0]); o

Matrice particolare char M[10][15]; o meglio

typedef char str[15];

str M[10];

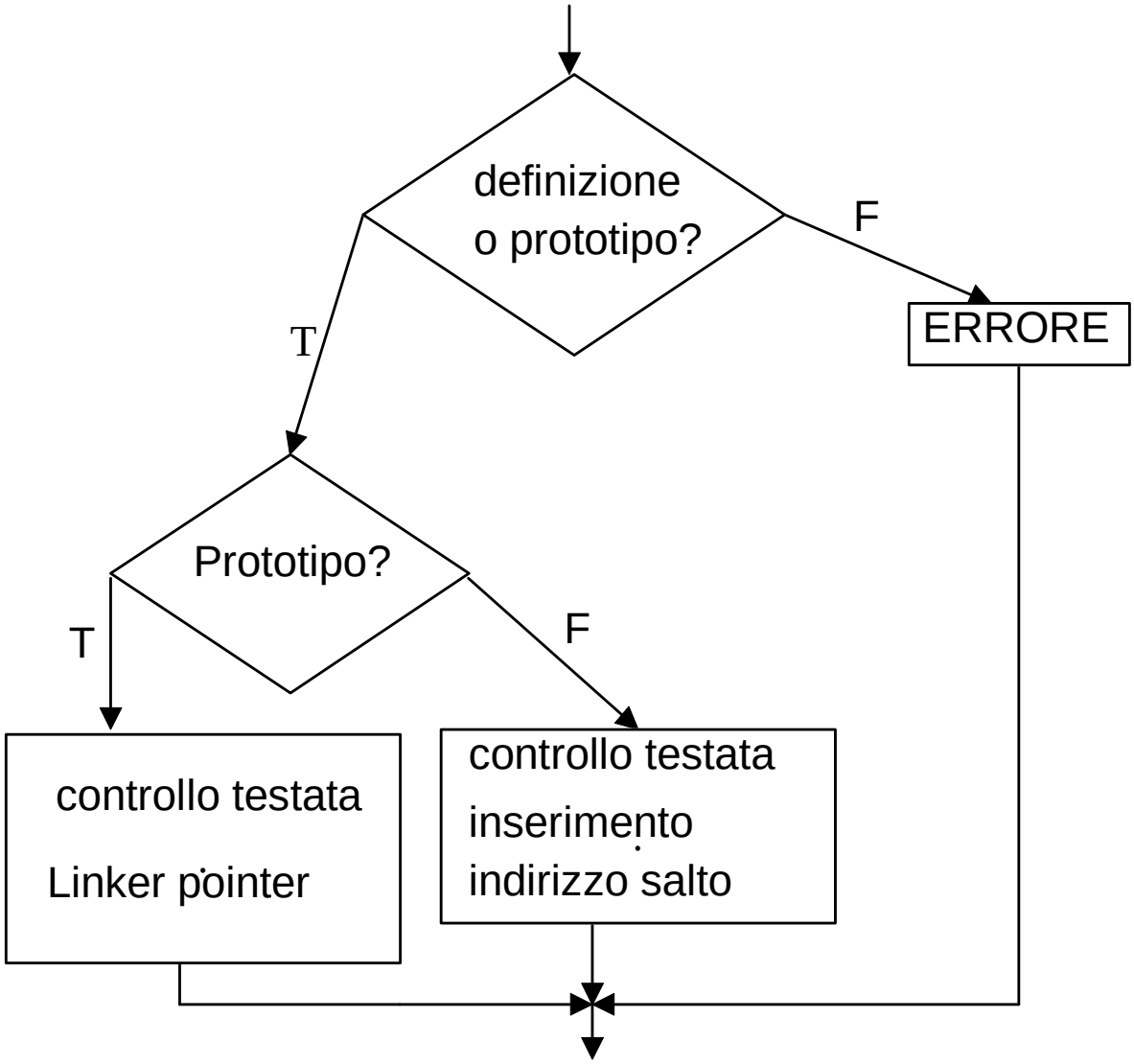
Passaggio parametri: f(str M[]) o f(str *M)

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(){ return(a+b); }
…
if (sum()>=low) printf ("sum=%d is over low", sum());
else printf("sum= %d is under low", sum());
```

Funzioni con effetti indesiderati

```
int a=2,b=1;
int sum()  { a++; return(a+b); }
…
if (sum()>=low) printf ("sum= %d is over low", sum());
else printf("sum= %d is under low", sum());
```

Funzioni con effetti desiderati

```
int progressivo=1000;
int insert(int val)
{//insert in file;
  progressivo++;
}
```

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;
void 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 ::= *constant_declarations* | *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 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*/
 }
```

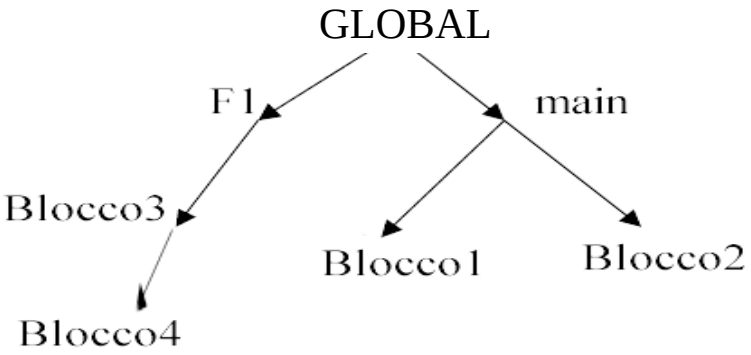
Definizioni:

- nomi globali ≡ nomi definiti in *global_declarative_part* incluso nomi di funzioni*;*
- nomi locali di un blocco ≡ nomi definiti nel blocco
- nomi locali di una funzione ≡ 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

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
Le istruzioni della funzione F1sono corrette?

1. SI locale a F1
2. NO improper use of typedef...
3. NO undefined symbol
4. SI globale

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

Variabili globali

- allocate staticamente dal compilatore a inizio esecuzione programma;
- tempo di vita ≡ 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≡ 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

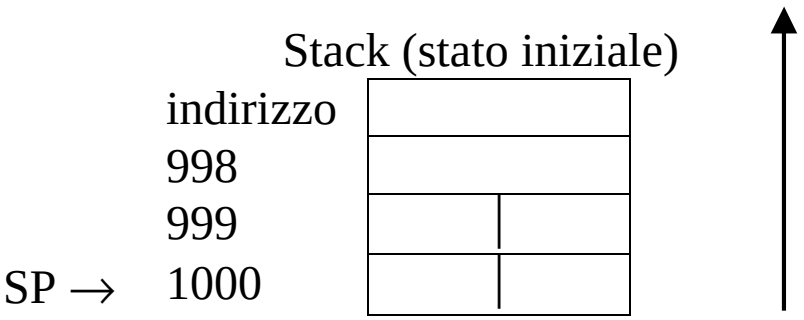
Il record di attivazione (RDA)

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 ≡ 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 all'atto della invocazione di sum — SP->

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
```

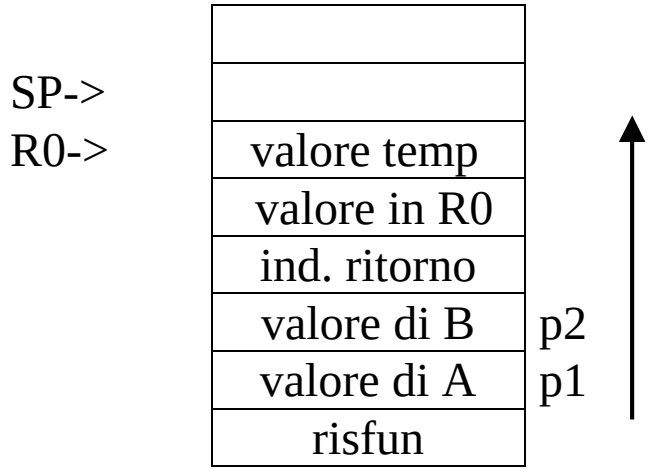
Cosa accade quando il main esegue l'istruzione JTS:

- 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)!

⇓

.....

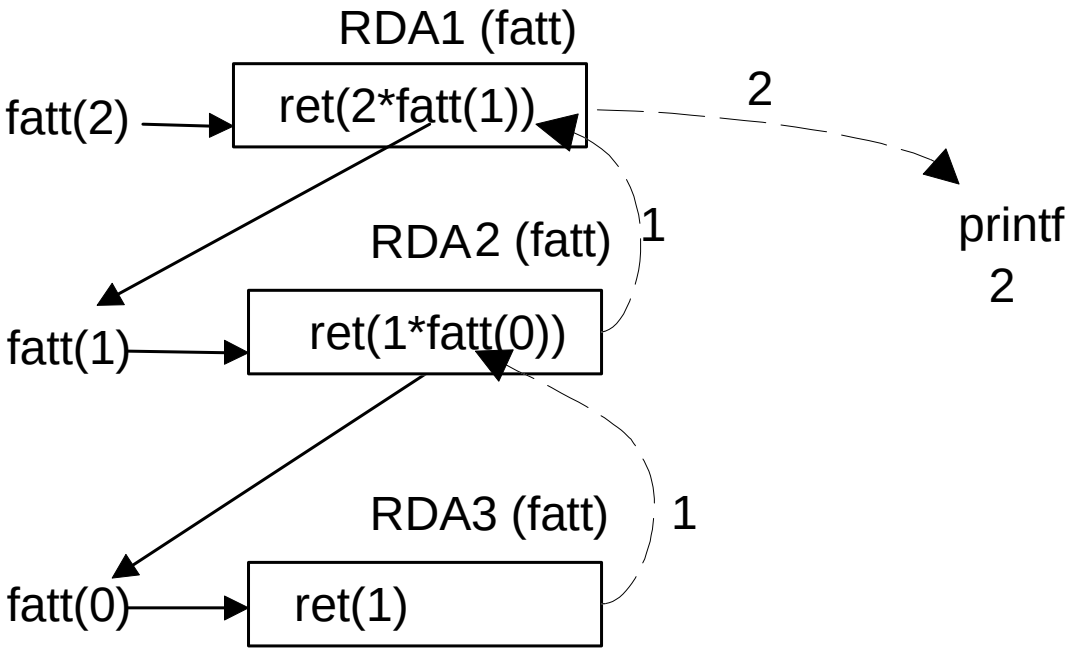
```
/*Soluzione ricorsiva diretta*
#include <stdio.h>
int N,R;
int fatt(int n)
{if (n == 0)
      return(1);
  else return (n * fatt(n-1);}

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 ≡ alla soluzione del problema, ma su un insieme ridotto di dati.

Esecuzione ricorsiva per N=2



Esempio 2: serie di Fibonacci (modello di crescita)

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

$f_0 = 0$ (caso base)

$f_1 = 1$ (caso base)

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

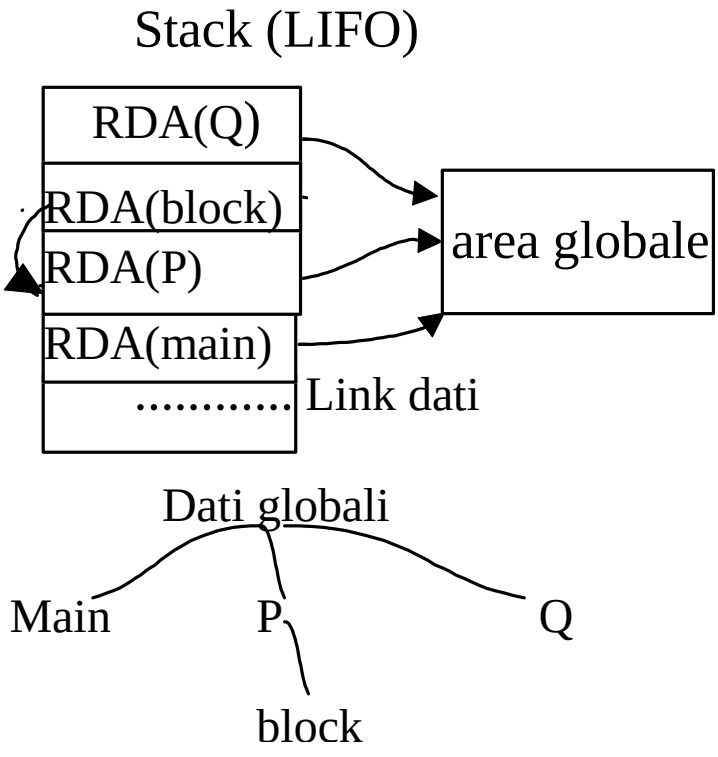
```
int   fibonacci (int  n)
{ int ultima,penultima, corrente, i;
  if (n==0) return 0;
  if (n==1) return 1;
  ultima=1; penultima=0;
  for(i=2; i<=n; i++)
       {corrente=ultima+penultima;
        penultima=ultima; ultima=corrente;
        }
   return (corrente);
}
```

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

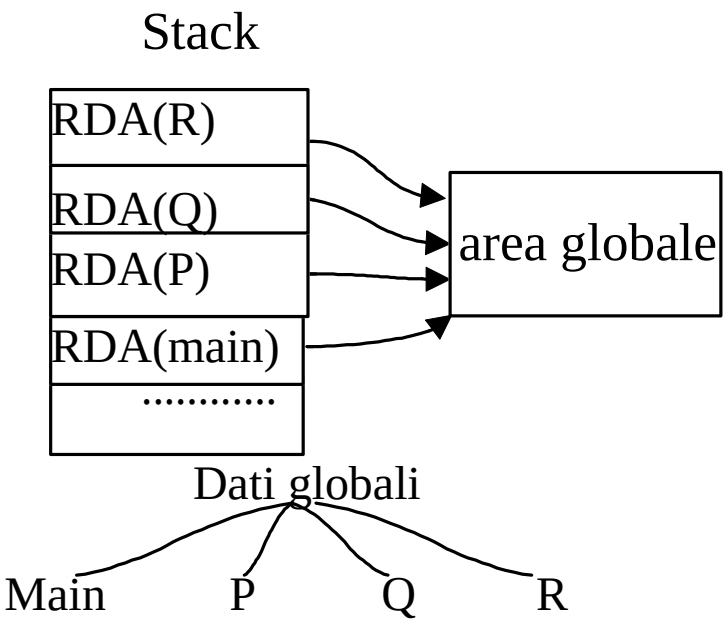
```
#include <stdio.h>
int    i = 1, max=100;
void sequenza ()
    { int numero; scanf ("%d",&numero);
      if (i==max)
          {printf("-%d",numero); return;}
      else
          {i++; sequenza(); printf("-%d",numero);}
     }
main() { sequenza();}
```

Stack e RDA con blocchi e ricorsione

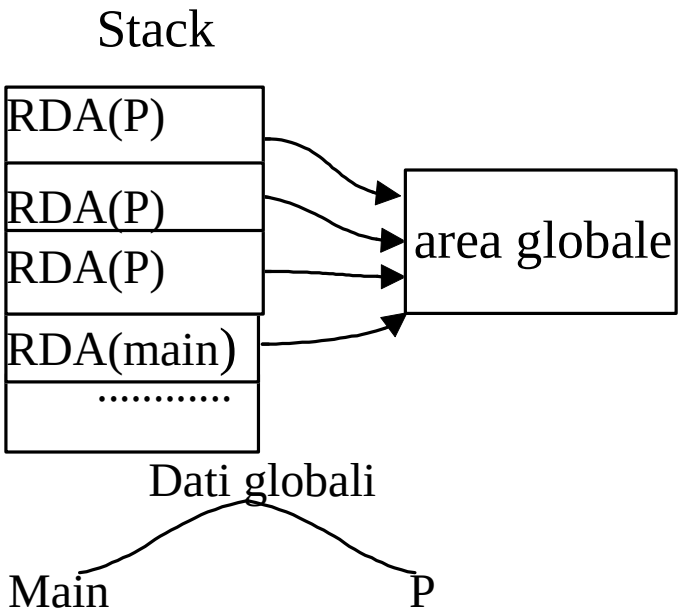
```
void main()
  {...P;}
void P()
  {…
    {/*block*/
      Q;
    }
  }
void Q()
  {...}
```



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




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



La gestione RDA con funzioni ricorsive

```
        ADD #-1, SP                                          R=fatt(N)
        MOV N, (SP)  ADD #-1, SP
        JSR fatt
ret:    ADD #+1, SP
        ADD #+1, SP  MOV (SP), R
        … (STOP)

fatt:   MOV R0, (SP)       ADD #-1, SP                       fatt (n)
        MOV SP, R0  ...
        MOV R0, R1   ADD #3, R1
        BREQ (R1), zero

ric:    MOV R0, R1   ADD #3, R1  MOV (R1), R1     n in R1
        ADD #-1, SP
       MOV  R0,  R3    ADD   #3,  R3    MOV  (R3),  (SP)   ADD  #-1,  (SP)
                        ADD #-1, SP       n-1 nello stack
        JSR fatt
Ret1:ADD #1, SP
        ADD #1, SP     MOV (SP), R2                  valore di fatt(n-1)

        //R2=R2*R1                                        n*fatt(n-1)
        MOV R0,R1  ADD#4, R1  MOV R2, (R1)
        BR fine

zero: MOV R0, R1  ADD #4, R1  MOV #1, (R1)

Fine:ADD #1, SP  MOV (SP), (R0)
        RTS
```

Puntatori a funzione

int (*f)() $\Rightarrow$ f contiene l'indirizzo (puntatore) di una funzione.

f() corrisponde ad invocare la funzione puntata da f

Es. int stampa(int a) {…}

```
main()
 { int (*f)();  f=&stampa; printf("%d", f(3));  }
```

Esempio più significativo (per meccanismo e per risultato): definire una funzione "derivata" che calcola la derivata di una funzione ricevuta come parametro (in questo esempio sarà $\sqrt{x}$) per approssimazioni di ε.

$$f'(x) = \frac{df(x)}{dx} = \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$

```
#include <stdio.h>
#include <math.h>
double radice(double x) {return sqrt(x);}

double derivata (double (*f)(double d), double x, double epsilon)
      {return ((f(x + epsilon)-f(x))/epsilon);}

void main()
     {double x, epsilon;   printf("x?");scanf("%lf",&x);
      epsilon=1.;
      while (epsilon > 1.e-17)
           {printf("\n x= %f, epsilon= %e,derivata=  %f", x, epsilon,
                    derivata(radice,x,epsilon));
            epsilon=epsilon/10.;
           }
      }
```

Sviluppo strutturato e incrementale di un'applicazione

- Dividi e conquista $\Rightarrow$ identificare le funzioni principali
- Le strutture dati globali
- Dall'algoritmo principale ai raffinamenti per passi
- Non puntare ad avere subito la soluzione ottima

Esempio:
Memorizzare i risultati di un test sostenuto da N studenti e visualizzare la media dei punteggi ottenuti dagli studenti suddivisi nelle tre classi di appartenenza (1, 2, 3).
Predisporre inoltre una operazione che permetta di ordinare i dati in base al numero di matricola.
Il programma contiene un menù con le seguenti voci: 1 per memorizzare i dati, 2 per la media, 3 per terminare l'esecuzione.

Lo schema di base del programma:
- una struttura dati globale;
- una funzione per il caricamento dati;
- una funzione per il calcolo della media;
- una funzione per l'ordinamento;
- la funzione main col menù.

Alcuni suggerimenti
- semplicità, chiarezza, niente trucchi (indice di ingenuità), massimizzare l'uso di parametri e dati locali;
- buona documentazione:
  - commenti all'inizio del programma per identificare nome del programmatore, data e versione del programma (vers. x.y), per descrivere obiettivi, strutture dati globali, algoritmi, funzioni utilizzate;
  - commenti nelle funzioni per descrivere obiettivi, algoritmi e strutture dati;
- stile:

- nomi significativi per gli identificatori (ad esempio: ImportoSalario);
- separare le istruzioni su linee diverse;
- utilizzare l'indentazione;
- spazi tra operandi di istruzioni/espressioni, linee bianche di separazione tra segmenti diversi di codice.

La struttura dati:

```
#define N 100            /*max numero di studenti */

typedef struct {int matricola, punteggio,classe;} T_studente;
typedef T_studente T_arc[N];
T_arc archivio;

int ultimo = -1;        /*gestione dinamica del vettore archivio*/
```

La funzione main

```
/* Program:… Responsabile:… data:… vers. 1.0 */

/*aggiunti alle strutture dati*/
#include <stdio.h>
typedef enum{False, True} boolean;

void main()
{ int scelta; boolean fine=False;
  while (!fine)
  { printf("\n1x inserisci, 2 x media, 3 x fine: ");
    scanf ("%d",&scelta);
    switch (scelta)
    {case 1: printf("memorizza i dati"); break;
      case 2: printf("calcolo media"); break;
      case 3: fine =True; /*fine*/
     }
  }
}
```

Sviluppo funzioni come stubs:

```
void MemorizzaDati(T_studente *s, int MAX, int *u)
{ printf("\nmemorizza dati");
/* 1. chiede il numero di studenti
2. per ogni studente legge i dati
3. restituisce il vettore e la posizione dell'ultimo elemento caricato
*/
}

int CalcoloMedia(T_studente *s, int u),
{printf("\nCalcoloMedia"); return(0); }

void Ordina(T_studente *s, int u),
{printf("\nOrdina"); }
```

Raffinamento funzioni:

```
void MemorizzaDati(T_studente *s, int MAX, int *u);
{int quanti; boolean ok=False; int i;
 /* 1. chiede il numero di studenti*/
 do
  { printf("\nquanti studenti? ");
    if (scanf("%d",&quanti) ==1)
     {if ((quanti >=0) &&(quanti<= MAX)) ok=True;}
  }
 while (!ok);

/*2. per ogni studente legge i dati*/
    for (i=0;i<quanti;i++)
     {printf("\nmatricola: ");scanf("%d",& (s+i)->matricola);
      printf("\npunteggio: ");scanf("%d",& (s+i)->punteggio);
      printf("\nclasse: ");scanf("%d",& (s+i)->classe); ≡ &s[i].classe
      }
```

```
/*3. restituisce il vettore e l'indice dell'ultimo elemento caricato*/
*u = quanti -1;
}

float CalcoloMedia(T_studente *s, int u)
{int i; float somma=0.0;
  for (i=0;i<=u;i++) somma = somma + (s+i)->punteggio;
  return(somma/(u+1));
 }
```

La verifica dell'applicazione (funzionale o prestazionale?)

Verifica funzionale:

- Verificare le singole funzioni del programma; un programma non sempre utilizza tutte le funzionalità offerte (ad esempio, la funzione Ordina);
- il test:
  - Si procede al test supportati da eventuale funzione di stampa;

/*Procedura di test*/

```
void Stampa (T_studente *s, int u)
{   int i;
    for (i=0;i<=u;i++)
    {printf("\nmatricola: %d", (s+i)->matricola);
     printf("\npunteggio: %d",(s+i)->punteggio);
     printf("\nclasse: %d", (s+i)->classe);
     }
}
```

- identificare il tipo di ingresso da verificare: grandi o piccoli insiemi, valori limite, dati errati, particolari ordinamenti dei dati; ad esempio, inserire 0, N e N+1 elementi nel vettore, errori di tipo, violazione di range o overflow in lettura);
- identificare le sequenze di attivazione delle procedure (ordinamento sequenziale o casuale nelle menù driven);
- identificare il tratto critico da sottoporre a verifica (ad esempio, calcoli nei quali si può generare overflow i++, N/0); identificare le variabili da verificare, copertura delle istruzioni;
- definire lo strumento di controllo (debugger o istruzioni di write).