

{
 ||
 ||
 ||
 || }
 NAME → EFFETTO
 |
 VALORE

Funzioni

ASTRAZIONE
PROCEDURALE
Marco Alberti

||
 ||
 ||
 || }
 NAME



Dipartimento
di Matematica
e Informatica



Università
degli Studi
di Ferrara

Programmazione e Laboratorio, A.A. 2020-2021

Ultima modifica: 23 ottobre 2020

Attenzione! Questo materiale didattico è per uso personale dello studente ed è coperto da copyright.
Ne sono vietati la riproduzione e il riutilizzo anche parziale, ai sensi e per gli effetti della legge sul diritto d'autore.

Sommario

1 Funzioni

- Subroutine .
- Parametri .
- Valore di ritorno .

2 Modello a run-time delle funzioni

3 Campo di visibilità (scope) degli identificatori

a

a

a

Sommario

1 Funzioni

- Subroutine
- Parametri
- Valore di ritorno

2 Modello a run-time delle funzioni

3 Campo di visibilità (scope) degli identificatori

Sommario

1 Funzioni

- Subroutine
- Parametri
- Valore di ritorno

2 Modello a run-time delle funzioni

3 Campo di visibilità (scope) degli identificatori

Rubrica con logo

Supponiamo di voler stampare un logo prima di ogni messaggio della nostra rubrica.

070_funzioni/logo.c

```
1 #include <stdio.h>
2
3 main() {
4     int scelta;
5     printf("*** Rubrica ***\n");
6     printf("1. Tizio 835-1234567\n");
7     printf("2. Caio 347-1234567\n");
8     printf("Scegliere un contatto per toccare\n");
9     scanf("%d", &scelta);
10    if (scelta > 0) {
11        printf("*** Rubrica ***\n");
12        printf("Chiamata del contatto %d in corso\n", scelta);
13    }
14    printf("Fine\n");
15 }
```

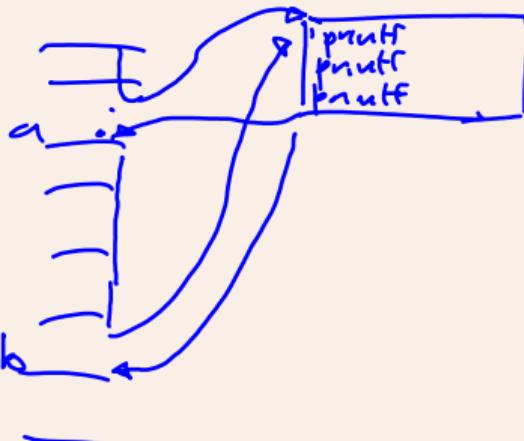
Se volessimo modificare il logo...

070_funzioni/logo2.c

```

1 #include <stdio.h>
2
3 main() {
4     int scelta;
5     printf("#####\n");
6     printf("# Rubrica #\n");
7     printf("#####\n");
8     printf("1. Tizio 335-1234567\n");
9     printf("2. Caio 347-1234567\n");
10    printf("Scegliere un contatto\n");
11    scanf("%d", &scelta);
12    if (scelta > 0) {
13        printf("#####\n");
14        printf("# Rubrica #\n");
15        printf("#####\n");
16        printf("Chiamata del contatto %d in corso\n", scelta);
17    }
18    printf("Fine\n");
19 }

```



Pattern subroutine (o sottoprocedura)

Abbiamo bisogno di eseguire due volte lo stesso codice, che si ripete alle righe 5-7 e 13-15.

In questo caso l'iterazione non aiuta, perché il codice deve eseguito non consecutivamente.

Possiamo scrivere il codice una volta sola e *chiamarlo* due volte.

Che cosa significa chiamare una porzione di codice (detta sottoprocedura o subroutine)?

- ① Saltare al codice GOTO
- ② Eseguirlo
- ③ Una volta eseguito, ritornare al punto di chiamata

Quindi l'ultima istruzione della sottoprocedura deve essere un salto al punto di chiamata, che deve essere identificato in qualche modo.

070_funzioni/subroutine-goto.c

```
1 #include <stdio.h>
2 main() {
3     int scelta, ritorno;
4     ritorno = 1; goto logo;
5 elenco:
6     printf("1. Tizio 335-1234567\n");
7     printf("2. Caio 347-1234567\n");
8     printf("Scegliere un contatto\n");
9     scanf("%d", &scelta);
10    ritorno = 2; goto logo;
11 chiamata:
12    printf("Chiamata del contatto %d in corso\n", scelta);
13    goto fine;
14 logo: SUBROUTINE
15    printf("#####\n");
16    printf("# Rubrica #\n");
17    printf("#####\n");
18    if (ritorno == 1) goto elenco;
19    else if (ritorno == 2) goto chiamata;
20 fine:
21    printf("Fine\n");
22 }
```

PUNTO DI RITORNO

CHIAMATA

SUBROUTINE

Osservazioni

- Il programma alla slide 4 (con subroutine) è più lungo di quello alla slide 2 con ripetizione...
- Ma il codice ripetuto è molto breve per far stare tutto il programma in una slide. Se anziché tre righe fosse 300, o se anziché essere chiamato 2 volte fosse chiamato 200 volte, il vantaggio sarebbe evidente.
- Ma anche per subroutine molto brevi o richiamate poche volte, evitare la ripetizione migliora la manutenibilità del programma. Inoltre comunica a chi legge che la funzionalità richiesta a ogni chiamata è effettivamente la stessa.

Funzioni

- Il pattern subroutine è una tecnica fondamentale per la strutturazione dei programmi e per evitare la duplicazione di codice.
- Per il programmatore è un po' scomodo gestire manualmente il punto di ritorno (oltre alle altre caratteristiche del pattern che vedremo) usando variabili apposite.
- Per questo (quasi) ogni linguaggio di programmazione contiene uno o più costrutti per implementarlo senza utilizzo esplicito di salti e gestendo automaticamente il punto di ritorno (e le altre caratteristiche che vedremo).
- In C questo costrutto è detto **funzione**.

Funzioni in C (versione provvisoria)

Definizione e chiamata di funzione

- $\langle \text{definizioneDiFunzione} \rangle ::= \langle \text{identificatore} \rangle () \langle \text{blocco} \rangle$

NOME { ... }

logo

$\langle \text{chiamataDiFunzione} \rangle ::= \langle \text{identificatore} \rangle ()$

logo

$\langle \text{identificatore} \rangle$ è il **nome** della funzione. $\langle \text{blocco} \rangle$ contiene la subroutine.

CHIAMATA DI FUNZIONE *logo()*

Semantica dell'espressione __ nome __()

- ① esegui il blocco della funzione __ nome __;
- ② riprendi dal punto di chiamata.

La definizione della funzione, in sè, non fa nulla; serve solo a definire cosa fare in caso di chiamata.

Vantaggio rispetto al pattern subroutine: è la macchina astratta a ricordarsi il punto di chiamata, quindi il programmatore non deve implementare esplicitamente il ritorno del flusso al punto di chiamata (per mezzo di etichette, **goto** e variabile).

Funzione

070_funzioni/subroutine-funzione.c

```
1 #include <stdio.h>
2 INIZI
3 logo() {
4     printf("#####\n");
5     printf("# Rubrica #\n");
6     printf("#####\n");
7 }
8
9
10 main() {
11     int scelta;
12     INIZI
13     logo(); CHIAMATA
14     printf("1. Tizio 335-1234567\n");
15     printf("2. Caio 347-1234567\n");
16     printf("Scegliere un contatto\n");
17     scanf("%d", &scelta);
18     INIZI
19     logo(); CHIAMATA
20     printf("Chiamata del contatto %d in corso\n", scelta);
21
22     printf("Fine\n");
23 }
```

D
F

INIZI **CHIAMATA** **BLOCCO - SUBROUTINE**

Esercizio

Nel programma alla slide 8 individuare

- le definizioni di funzione;
- le chiamate di funzione.

Determinare inoltre il flusso (cioè l'elenco delle righe delle istruzioni eseguite).

Esercizio

Asterischi

Scrivere un programma che stampi 5 righe di 20 asterischi. In particolare, definire una funzione di nome **star** che stampi 20 asterischi con un ciclo **for**, e chiamarla 5 volte con un ciclo **for**.

```
star() {
    int i;
    for(i=0; i<20; i++) {
        printf("*");
    }
}
```

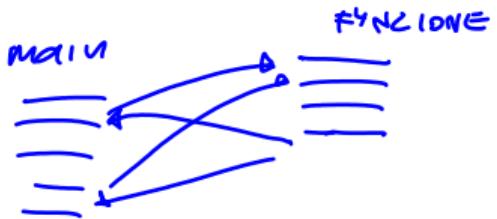
SUBROUTINE (FUNZIONE)

```
main() {
    int j;
    for(j=0; j<5; j++) {
        star();
    }
}
```

Sommario

1 Funzioni

- Subroutine
- Parametri
- Valore di ritorno



2 Modello a run-time delle funzioni

3 Campo di visibilità (scope) degli identificatori

Subroutine con parametri



- Supponiamo di voler stampare una volta il logo fatto di asterischi (*) e una volta fatto di cancelletti (#). Il pattern subroutine esegue sempre lo stesso codice, quindi pare che non possiamo utilizzarlo.
- Però, se il codice della subroutine contiene variabili, possiamo usarle per influenzare il comportamento della subroutine.
- Variabili utilizzate in questo modo sono dette **parametri**.
- Definizione di parametro del vocabolario Zingarelli: "grandezza che compare in un'espressione matematica o in una funzione, e il cui variare influenza altre variabili presenti o la natura della funzione o dell'ente matematico descritto dall'espressione".

Subroutine con parametri

070_funzioni/subroutine-goto-param.c

```
1 #include <stdio.h>
2
3 main() {
4     int scelta, ritorno, i; char carattere; PARAMETRO
5     ritorno = 1; carattere = '*'; goto logo;
6     elenco:
7     printf("1. Tizio 335-1234567\n");
8     printf("2. Caio 347-1234567\n");
9     printf("Scegliere un contatto\n");
10    scanf("%d", &scelta);
11    ritorno = 2; carattere = '#'; goto logo;
12    chiamata:
13    printf("Chiamata del contatto %d in corso\n", scelta);
14    goto fine;
15  logo:
16  {   for (i = 0; i < 15; i++)
17      printf("%c", carattere);
18      printf("\n");
19      printf("%c Rubrica %c\n", carattere, carattere);
20      for (i = 0; i < 15; i++)
21          printf("%c", carattere);
22      printf("\n");
23      if (ritorno == 1) goto elenco;
24      else if (ritorno == 2) goto chiamata;
25  fine:
26      printf("Fine\n");
27 }
```

Funzione con parametri

070_funzioni/subroutine-funzione-param.c

```

1 #include <stdio.h>
2 PARAMETRO FORMALE
3 logo(char carattere) {
4     int i;
5     for (i = 0; i < 15; i++)
6         printf("%c", carattere);
7     printf("\n");
8     printf("%c Rubrica %c\n", carattere, carattere);
9     for (i = 0; i < 15; i++)
10        printf("%c", carattere);
11    printf("\n");
12 }
13
14 main() {
15     int scelta;
16     PARAMETRO ATTUALE
17     logo(*);
18     printf("1. Tizio 335-1234567\n");
19     printf("2. Caio 347-1234567\n");
20     printf("Scegliere un contatto\n");
21     scanf("%d", &scelta);
22
23     logo('#');
24     printf("Chiamata del contatto %d in corso\n", scelta);
25
26     printf("Fine\n");
27 }

```

definizione
di funzione

chiamate

Funzioni con parametri in C (versione provvisoria)

Definizione e chiamata di funzione

$\langle \text{definizioneDiFunzione} \rangle ::= \langle \text{identificatore} \rangle (\langle \text{parametriFormali} \rangle) \langle \text{blocco} \rangle$
int a, char b
 $\langle \text{chiamataDiFunzione} \rangle ::= \langle \text{identificatore} \rangle (\langle \text{parametriAttuali} \rangle)$
int z, char z

$\langle \text{parametriFormali} \rangle ::= \langle \text{vuoto} \rangle$
 | **void**
 | $\langle \text{parametroFormale} \rangle [] , \langle \text{parametroFormale} \rangle]^*$

$\langle \text{parametroFormale} \rangle ::= \underbrace{\langle \text{tipo} \rangle}_{\text{char}} \underbrace{\langle \text{identificatore} \rangle}_{\text{Carattere}}$

$\langle \text{parametriAttuali} \rangle ::= \langle \text{vuoto} \rangle$
 | $\langle \text{espressione} \rangle [] , \langle \text{espressione} \rangle]^*$

$\langle \text{blocco} \rangle ::= \{ [] \langle \text{definizioneDiVariabile} \rangle]^* [] \langle \text{istruzione} \rangle]^* \}$

Funzioni con parametri in C

 v_1
↓ v_N
↓

Semantica della chiamata `__nome__(__exp1__, ..., __expN__)`

- ① assegna i valori di `__exp1__`, ..., `__expN__` ai parametri formali della funzione `__nome__` nelle rispettive posizioni (che quindi devono essere nello stesso numero e di tipo compatibile)
- ② esegue il blocco di `__nome__`;
- ③ riprende dal punto di chiamata.

Nel programma alla slide 13, alla prima chiamata della funzione `logo` al parametro formale `carattere` viene assegnato il parametro attuale `'*'`, e alla seconda chiamata il parametro attuale `'#'`. L'effetto delle due chiamate è quindi diverso.

Osservazioni

- I parametri formali funzionano come variabili inizializzate, a ogni chiamata della funzione, con i valori dei corrispondenti parametri attuali.
- Ogni chiamata di una funzione è completamente indipendente da qualsiasi altra (quindi, in particolare, i valori dei parametri formali e di eventuali variabili locali alla funzione non si mantengono fra una chiamata e l'altra).
- Vantaggi rispetto al pattern subroutine con parametri:
 - i parametri sono ben identificati e localizzati;
 - allo stesso modo, le variabili (locali) usate nella funzione sono definite nel blocco della funzione stessa, senza "sporcare" il resto del programma;
 - la macchina astratta gestisce da sola l'assegnamento dei parametri.

Esercizio

Nel programma alla slide 13 individuare

- le definizioni di funzione e i parametri formali di ognuna;
- le chiamate di funzione e i parametri attuali di ognuna.

Alla riga 10, qual è il valore di **carattere**?

Esercizio

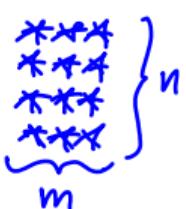
Rettangolo di asterischi

Scrivere un programma che stampi a video un rettangolo composto di asterischi di lati **m** e **n** richiesti all'utente, usando una funzione **riga** che stampi una riga di **m** asterischi.

INPUT

4 3
n m

OUTPUT



riga (int m) {

.. ..

}

main () {
for ()
riga (m);
}

Funzioni che chiamano funzioni

Nulla vieta che il corpo di una funzione contenga una chiamata a un'altra funzione. Ad esempio, nella funzione **logo** nella diapositiva 13 contiene due cicli **for** uguali. E' preferibile definire una funzione e chiamarla due volte.

```
riga(char ch){
```

```
    logo(char c){
```

```
        riga(c);
```

```
}
```

```
main(){
```

```
    logo('#');
```

```
}
```

Funzioni che chiamano funzioni

070_funzioni/subroutine-funzione-riga.c

```
1 #include <stdio.h>
2 quant quale
3 riga(int n, char carattere) {
4     int i;
5     for (i = 0; i < n; i++)
6         printf("%c", carattere);
7     printf("\n");
8 }
9 .
10 logo(char carattere) {
11     riga(15, carattere);
12     printf("%c Rubrica %c\n", carattere, carattere);
13     riga(15, carattere);
14 }
15
16 main() {
17     int scelta, ritorno;
18     logo('*');
19     printf("1. Tizio 335-1234567\n");
20     printf("2. Caio 347-1234567\n");
21     printf("Scegliere un contatto\n");
22     scanf("%d", &scelta);
23     logo('#');
24     printf("Chiamata del contatto %d in corso\n", scelta);
25     printf("Fine\n");
26 }
```

Sommario

1 Funzioni

- Subroutine
- Parametri
- Valore di ritorno

2

2 Modello a run-time delle funzioni

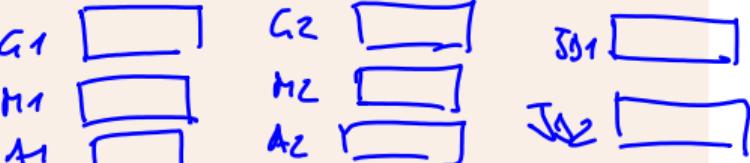
3 Alcuni esercizi

4 Campo di visibilità (scope) degli identificatori

Ripasso: differenza fra date con giorno juliano

070_funzioni/differenza-fra-date.c

```
1 #include <stdio.h>
2 main() {
3     int G, M, A, G1, M1, A1, G2, M2, A2;
4     int N0, N1, N2, N3; int JD1, JD2;
5     printf("Prima data?\n"); scanf("%d%d%d", &G1, &M1, &A1);
6     N0 = (M1 - 14) / 12;
7     N1 = 1461 * (A1 + 4800 + N0) / 4;
8     N2 = 367 * (M1 - 2 - 12 * N0) / 12;
9     N3 = 3 * (A1 + 4900 + N0) / 400;
10    JD1 = N1 + N2 - N3 + G1 - 32075; GIORNO GIULIANO PRIMA DATA
11    printf("Seconda data?\n"); scanf("%d%d%d", &G2, &M2, &A2);
12    N0 = (M2 - 14) / 12;
13    N1 = 1461 * (A2 + 4800 + N0) / 4;
14    N2 = 367 * (M2 - 2 - 12 * N0) / 12;
15    N3 = 3 * (A2 + 4900 + N0) / 400;
16    JD2 = N1 + N2 - N3 + G2 - 32075; GIORNO GIULIANO SECONDA DATA
17    printf("La differenza e' %d\n", JD2 - JD1);
18 }
```



RISULTATI
INTERMEDI

RISULTATI
INTERMEDI

Utilizzo di subroutine per calcoli

- Finora abbiamo usato le subroutine (realizzate con o senza funzioni) per ottenere più di un effetto, eventualmente dipendente da parametri, scrivendo il codice una volta sola.
- Succede spesso di voler eseguire lo stesso procedimento di calcolo su dati diversi.
- Anche a questo scopo si possono usare le subroutine, usando una variabile per memorizzare il risultato del calcolo e utilizzarlo al ritorno dalla subroutine.
- le funzioni supportano anche questa caratteristica del pattern subroutine.



Differenza fra date con subroutine

070_funzioni/differenza-fra-date-goto.c

```
1 #include <stdio.h>
2 main() {
3     int G, M, A, G1, M1, A1, G1, M2, A2;
4     int N0, N1, N2, N3; int JD, JD1, JD2;
5     int ritorno;
6     printf("Prima data?\n"); scanf("%d%d%d", &G1, &M1, &A1);
7     ritorno = 1; G = G1; M = M1; A = A1; goto calcolo;
8 r1:
9     JD1 = JD;
10    printf("Seconda data?\n"); scanf("%d%d%d", &G2, &M2, &A2);
11    ritorno = 2; G = G2; M = M2; A = A2; goto calcolo;
12 r2:
13     JD2 = JD;
14     goto fine;
15 calcolo:
16     N0 = (M - 14) / 12;
17     N1 = 1461 * (A + 4800 + N0) / 4;
18     N2 = 367 * (M - 2 - 12 * N0) / 12;
19     N3 = 3 * (A + 4900 + N0) / 400;
20     JD = N1 + N2 - N3 + G - 32075;
21     if (ritorno == 1)
22         goto r1;
23     else if (ritorno == 2)
24         goto r2;
25 fine:
26     printf("La differenza e' %d\n", JD2 - JD1);
27 }
```

Annotations in blue:

- Line 1: **PARAMETRI** → **VALORE DI RITORNO**
- Line 3: **PUNTO DI RITORNO**
- Line 16-20: **SUBROUTINE**

Differenza fra date con funzione

070_funzioni/differenza-fra-date-funzione.c

```

    → TIPO DI RITORNO
1 #include <stdio.h>
2 int giorno_giuliano(int g, int m, int a) {
3     int N0, N1, N2, N3;
4     N0 = (m - 14) / 12;
5     N1 = 1461 * (a + 4800 + N0) / 4;
6     N2 = 367 * (m - 2 - 12 * N0) / 12;
7     N3 = 3 * (a + 4900 + N0) / 400;
8     return N1 + N2 - N3 + g - 32075;
9 }
      → VALORE DI RITORNO
10
11 main() {
12     int G1, M1, A1;
13     int G2, M2, A2;
14     int JD1, JD2;
15     printf("Prima data?\n"); scanf("%d%d%d", &G1, &M1, &A1);
16     JD1 = giorno_giuliano(G1, M1, A1); 11 2020
17     printf("Seconda data?\n"); scanf("%d%d%d", &G2, &M2, &A2);
18     JD2 = giorno_giuliano(G2, M2, A2);
19     printf("La differenza e' %d\n", JD2 - JD1);
20 }
  
```

Funzioni con parametri e valore di ritorno in C (versione definitiva (per ora))

Definizione e chiamata di funzione

$\langle definizioneDiFunzione \rangle ::= \langle tipo \rangle \langle identificatore \rangle (\langle parametriFormali \rangle) \langle blocco \rangle$

$\langle chiamataDiFunzione \rangle ::= \langle identificatore \rangle (\langle parametriAttuali \rangle)$

$\langle parametriFormali \rangle ::= \langle vuoto \rangle$

 | $\langle parametroFormale \rangle [, \langle parametroFormale \rangle]^*$

$\langle parametroFormale \rangle ::= \langle tipo \rangle \langle identificatore \rangle$

$\langle parametriAttuali \rangle ::= \langle vuoto \rangle$

 | $\langle espressione \rangle [, \langle espressione \rangle]^*$

$\langle istruzioneRitorno \rangle ::= \text{return} \langle espressione \rangle ;$

Valore di ritorno

- La semantica dell'istruzione `return __espr__;` è
 - assegnare all'espressione di chiamata il valore di `__espr__;`
 - far tornare il flusso al punto di chiamata.
- Se una funzione ha il tipo di ritorno, deve contenere una o più istruzioni di ritorno con espressione di quel tipo.
- Le funzioni senza valore di ritorno sono di tipo `void` (è bene indicarlo). Sono spesso dette `procedure` (altri linguaggi hanno costrutti separati per le procedure).
- Possono contenere istruzioni di ritorno senza espressione (`return;`), non necessariamente alla fine.

```
int f( ) {
    if ( . )
        return 1;
    else
        return 0;
}
```

`a = f(...);`
 ↓
 VAORE
 int f(...)
 {
 return;
 }

```
void logo (char c) {
    }
}
```

Utilizzo diretto di funzioni in espressione

070_funzioni/differenza-fra-date-funzione-sottrazione.c

```

1 #include <stdio.h>
2 int giorno_giuliano(int g, int m, int a) {
3     int N0, N1, N2, N3;
4     N0 = (m - 14) / 12;
5     N1 = 1461 * (a + 4800 + N0) / 4;
6     N2 = 367 * (m - 2 - 12 * N0) / 12;
7     N3 = 3 * (a + 4900 + N0) / 400;
8     return N1 + N2 - N3 + g - 32075;
9 }
10
11 main() {
12     int G1, M1, A1;
13     int G2, M2, A2;
14     G1, M1, A1, G2, M2, A2
15     printf("Prima data?\n"); scanf("%d%d%d", &G1, &M1, &A1);
16     printf("Seconda data?\n"); scanf("%d%d%d", &G2, &M2, &A2);
17     printf("La differenza e` %d\n", - giorno_giuliano(G1, M1, A1) -
18                                     - giorno_giuliano(G2, M2, A2));
19 }
```

$(2+5) - (4+1)$

Interfaccia di una funzione

- $\langle tipo \rangle \langle identificatore \rangle (\langle parametriFormali \rangle)$ (cioè la definizione di una funzione senza il corpo) è detto **prototipo**, interfaccia o **firma** (**signature**).
- La chiamata di funzione è un'espressione con un tipo (uguale al tipo di ritorno della funzione), e può essere usata in altre espressioni (ad esempio, data l'interfaccia `int f(void)`, 3 + f() è un'espressione intera valida). f(ers)
- Il compilatore controlla che ogni chiamata di una funzione ne rispetti l'interfaccia (cioè che numero e tipo dei parametri attuali corrispondano a quelli dei parametri formali, e che il tipo di ritorno sia compatibile con l'espressione in cui è usata la chiamata).

Esempio

070_funzioni/potenza.c

```

1 #include <stdio.h>
2     INTERFACE
3 int potenza(int base, int esp) {
4     int cont, prod = 1;
5     for (cont = 0; cont < esp; cont++)
6         prod *= base;
7     return prod;
8 }
9
10 main() {
11     int a = 3;
12     int b = 2;
13     printf("%d\n", potenza(a, b));
14     printf("%d\n", potenza(b, a));
15 }
```

definizione di
funzione

a [3]

b [2]

3

2

3

8

Che cosa fa questo programma?

Potenza

```
1 #include <stdio.h>
2
3 int potenza(int base, int esp) {
4     int cont, prod = 1;
5     for (cont = 0; cont < esp; cont++)
6         prod *= base;
7     return prod;
8 }
9
10 main() {
11     int a = 3, b = 2;
12     printf("%d\n", potenza(a, b));
13     printf("%d\n", potenza(b, a));
14 }
15 }
```

a	4294951772
3	
b	4294951768
2	

main (13)

Potenza

```
1 #include <stdio.h>
2
3 int potenza(int base, int esp) {
4     int cont, prod = 1;
5     for (cont = 0; cont < esp; cont++) accumulatore
6         prod *= base;
7     return prod;
8 }
9
10 main() {
11     int a = 3;
12     int b = 2;
13     printf("%d\n", potenza(a, b));
14     printf("%d\n", potenza(b, a));
15 }
```

reduce (*, 1, [3, 3])

main (13)	potenza (5)
a	4294951772
b	4294951768
base	4294951752
esp	4294951756
cont	136254341
prod	4294951736

Potenza

```
1 #include <stdio.h>
2
3 int potenza(int base, int esp) {
4     int cont, prod = 1;
5     for (cont = 0; cont < esp; cont++)
6         prod *= base;
7     return prod;
8 }
9
10 main() {
11     int a = 3;
12     int b = 2;
13     printf("%d\n", potenza(a, b));
14     printf("%d\n", potenza(b, a));
15 }
```

a	4294951772
b	3
base	4294951768
esp	2
cont	4294951752
prod	0

main (13) |
potenza (6)

Potenza

```
1 #include <stdio.h>
2
3 int potenza(int base, int esp) {
4     int cont, prod = 1;
5     for (cont = 0; cont < esp; cont++)
6         prod *= base;
7     return prod;
8 }
9
10 main() {
11     int a = 3;
12     int b = 2;
13     printf("%d\n", potenza(a, b));
14     printf("%d\n", potenza(b, a));
15 }
```

main (13)	potenza (6)
a	4294951772 3
b	4294951768 2
base	4294951752 3
esp	4294951756 2
cont	4294951740 1
prod	4294951736 3

Potenza

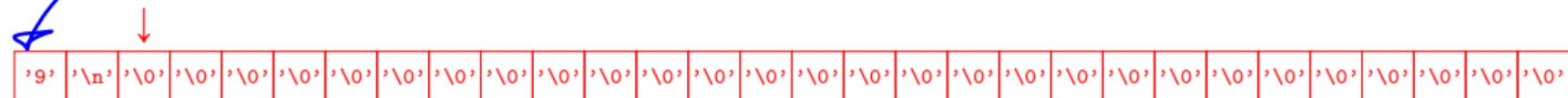
```
1 #include <stdio.h>
2
3 int potenza(int base, int esp) {
4     int cont, prod = 1;
5     for (cont = 0; cont < esp; cont++)
6         prod *= base; 2 2
7     return prod; 7
8 }
9
10 main() {
11     int a = 3;
12     int b = 2;
13     printf("%d\n", potenza(a, b)); 13
14     printf("%d\n", potenza(b, a));
15 }
```

a	4294951772
b	4294951768
base	4294951752
esp	4294951756
cont	4294951740
prod	4294951736

main (13) |
potenza (7)

Potenza

```
1 #include <stdio.h>
2
3 int potenza(int base, int esp) {
4     int cont, prod = 1;
5     for (cont = 0; cont < esp; cont++)
6         prod *= base;
7     return prod;
8 }
9
10 main() {
11     int a = 3;
12     int b = 2;
13     printf("%d\n", potenza(a, b));
14     printf("%d\n", potenza(b, a));
15 }
```



stdout

main(14)

Potenza

```
1 #include <stdio.h>
2
3 int potenza(int base, int esp) {
4     int cont, prod = 1;
5     for (cont = 0; cont < esp; cont++)
6         prod *= base;
7     return prod;
8 }
9
10 main() {
11     int a = 3;
12     int b = 2;
13     printf("%d\n", potenza(a, b));
14     printf("%d\n", potenza(b, a));
15 }
```

ACCUMULATORE

stdout

a	4294951772	3
b	4294951768	2
base	4294951744	2
esp	4294951748	3
cont	134227632	4294951732
prod	4294951728	1

Potenza

```
1 #include <stdio.h>
2
3 int potenza(int base, int esp) {
4     int cont, prod = 1;
5     for (cont = 0; cont < esp; cont++)
6         prod *= base;
7     return prod; 2
8 }
9
10 main() {
11     int a = 3;
12     int b = 2;
13     printf("%d\n", potenza(a, b));
14     printf("%d\n", potenza(b, a));
15 }
```



stdout

a	4294951772
b	4294951768
base	4294951744
esp	4294951748
cont	4294951732
prod	4294951728

3
2
2
3
0
1

main (14)
potenza (6)

Potenza

```
1 #include <stdio.h>
2
3 int potenza(int base, int esp) {
4     int cont, prod = 1;
5     for (cont = 0; cont < esp; cont++)
6         prod *= base;
7     return prod;
8 }
9
10 main() {
11     int a = 3;
12     int b = 2;
13     printf("%d\n", potenza(a, b));
14     printf("%d\n", potenza(b, a));
15 }
```



stdout

a	4294951772
b	3
base	4294951768
esp	2
cont	4294951744
prod	3
	4294951732
	1
	4294951728
	2

Potenza

```
1 #include <stdio.h>
2
3 int potenza(int base, int esp) {
4     int cont, prod = 1;
5     for (cont = 0; cont < esp; cont++)
6         prod *= base;
7     return prod;
8 }
9
10 main() {
11     int a = 3;
12     int b = 2;
13     printf("%d\n", potenza(a, b));
14     printf("%d\n", potenza(b, a));
15 }
```

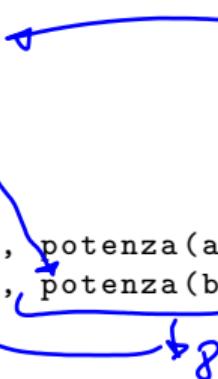


stdout

a	4294951772	3
b	4294951768	2
base	4294951744	2
esp	4294951748	3
cont	4294951732	2
prod	4294951728	4

Potenza

```
1 #include <stdio.h>
2
3 int potenza(int base, int esp) {
4     int cont, prod = 1;
5     for (cont = 0; cont < esp; cont++)
6         prod *= base;
7     return prod;
8 }
9
10 main() {
11     int a = 3;
12     int b = 2;
13     printf("%d\n", potenza(a, b));
14     printf("%d\n", potenza(b, a));
15 }
```



stdout

a	4294951772
b	3
base	4294951768
esp	2
cont	4294951744
prod	4294951748
	3
	4294951732
	3
	4294951728
	8

main (14)
potenza (7)

8\n

Potenza

```

1 #include <stdio.h>
2
3 int potenza(int base, int esp) {
4     int cont, prod = 1;
5     for (cont = 0; cont < esp; cont++)
6         prod *= base;
7     return prod;
8 }
9
10 main() {
11     int a = 3;
12     int b = 2;
13     printf("%d\n", potenza(a, b));
14     printf("%d\n", potenza(b, a));
15 }
```

a	4294951772
3	
b	4294951768
2	

main (15)

'9'	'\n'	'8'	'\n'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'
-----	------	-----	------	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

stdout

Tipo della funzione `main`

```
int main() {  
    if (...) return 1;  
    return 0; }  
}
```

- `main` è una funzione! E il suo tipo è `int`.
- L'esecuzione del programma comincia sempre dalla funzione `main`. In modo un po' impreciso, si può dire che, quando invochiamo file eseguibile ottenuto dalla compilazione di un programma, il sistema operativo chiama la funzione `main` di quel programma.
- Il valore di ritorno è usato per comunicare al sistema operativo l'esito dell'esecuzione del programma: convenzionalmente, `0` significa che non ci sono stati problemi, mentre altri valori indicano errori di tipo dipendente dall'applicazione.

Quindi d'ora in poi scriveremo i programmi così:

```
int main(void){  
    ...  
    return 0; // tutto ok  
}
```

Sommario

1 Funzioni

- Subroutine
- Parametri
- Valore di ritorno

2 Modello a run-time delle funzioni

RECORD DI ATTIVAZIONE



3 Alcuni esercizi

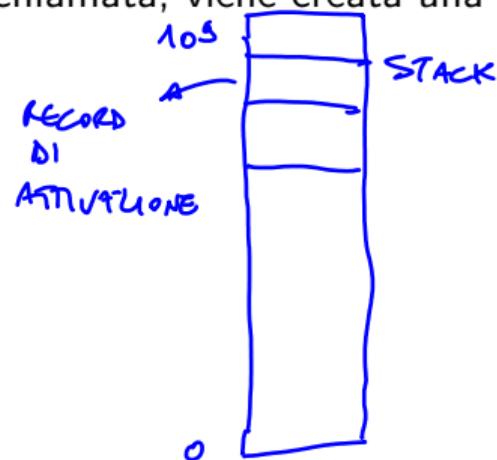
4 Campo di visibilità (scope) degli identificatori

Gestione delle chiamate di funzioni

Abbiamo visto che, tramite le funzioni, la macchina astratta gestisce autonomamente questi elementi del pattern procedura:

- punto di ritorno
- parametri
- valore di ritorno

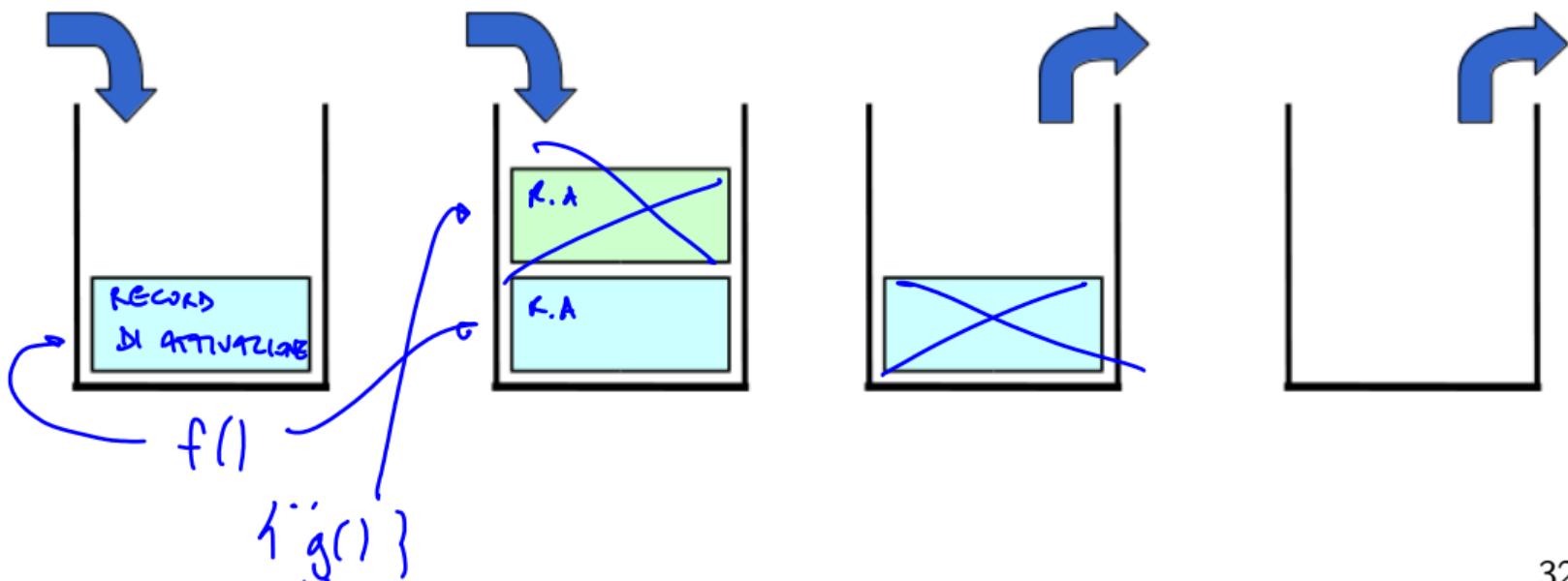
La macchina astratta usa un'area di memoria apposita (detta **stack**) dove, per ogni chiamata, viene creata una struttura di dati detta **record di attivazione** o **stack frame**.



Pila (o stack) dei record di attivazione

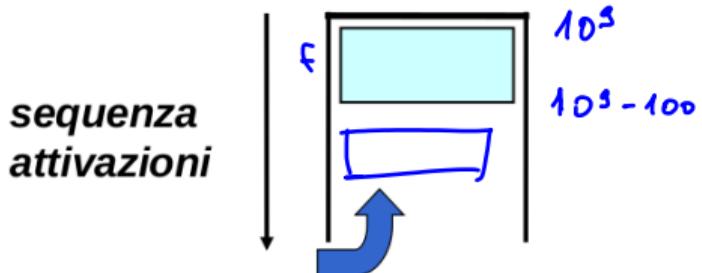
Ad ogni chiamata di funzione, viene creato un record di attivazione che verrà distrutto alla fine dell'esecuzione della funzione. In questo modo l'ultimo record creato è anche il primo a essere eliminato (regola LIFO - Last In First Out).

Un modo per implementare questa politica è caricare ogni record di attivazione sopra quello del chiamante, formando una pila; i record vengono rimossi dalla cima della pila.

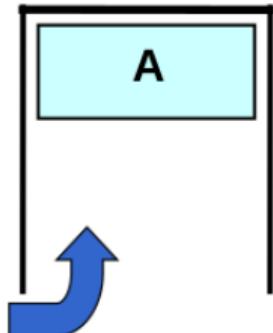


Sequenza di attivazioni

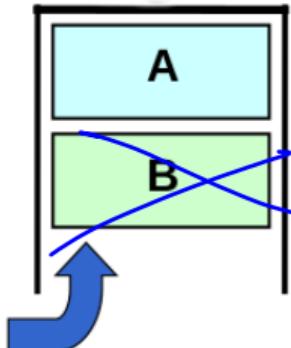
In realtà lo stack cresce dagli indirizzi alti a quelli bassi, quindi lo stack si disegna solitamente così:



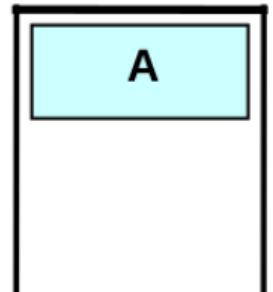
Quindi se la funzione **A** chiama la funzione **B**:



**A chiama B
e passa il
controllo a B**



**poi B
finisce e il
controllo
torna ad A**



Record di attivazione

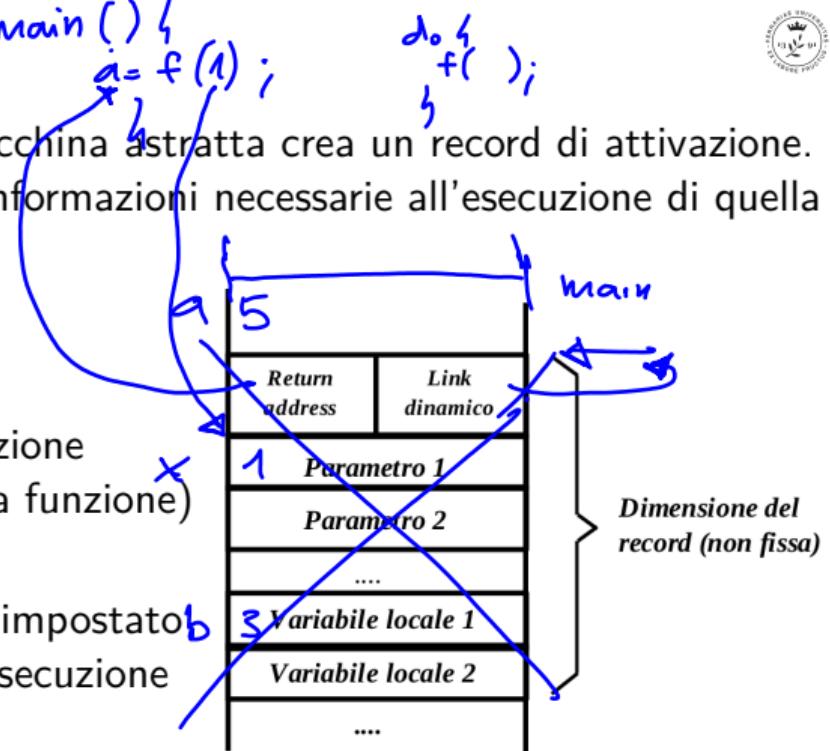
```

int f(int x) {
    int b;
    b = x;
    return b;
}

main() {
    a = f(1);
    do {
        f();
    }
}
    
```

Per ogni chiamata di ogni funzione la macchina astratta crea un record di attivazione. Il record di attivazione contiene tutte le informazioni necessarie all'esecuzione di quella chiamata di funzione:

- parametri
- variabili locali
- punto di ritorno (l'indirizzo dell'istruzione macchina da eseguire al termine della funzione)
- link dinamico (indirizzo del record di attivazione del chiamante, che verrà impostato come cima dello stack alla fine dell'esecuzione della funzione)

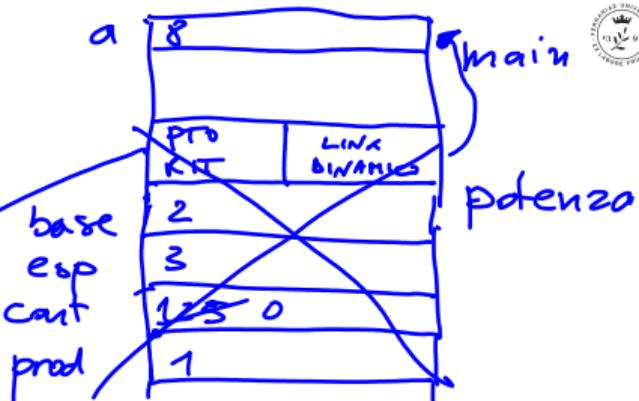


Chiamate diverse alla stessa funzione generano record di attivazione diversi.

Esempio di stack frame

Data la seguente definizione:

```
int potenza(int base, int esp) {
    int cont, prod = 1;
    for (cont = 0; cont < esp; cont++)
        prod *= base;
    return prod;
}
```



Il stack frame per la chiamata **potenza(2,3)** da parte della funzione **main** potrebbe essere questo:

```
main {
    int a;
    a = potenza(2,3);
    ...
}
```

Record di attivazione e valore di ritorno

Abbiamo visto come la macchina astratta, tramite i record di attivazione, gestisce due degli elementi del pattern sottoprocedura:

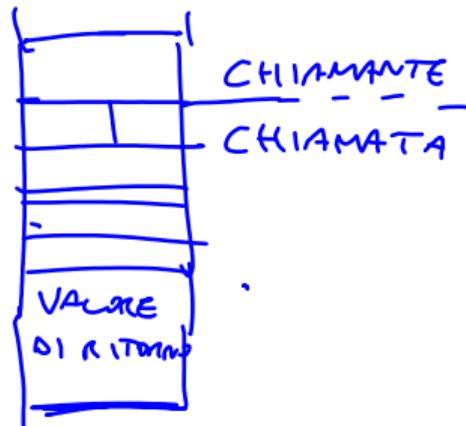
- punto di ritorno
- parametri

mov a, eax



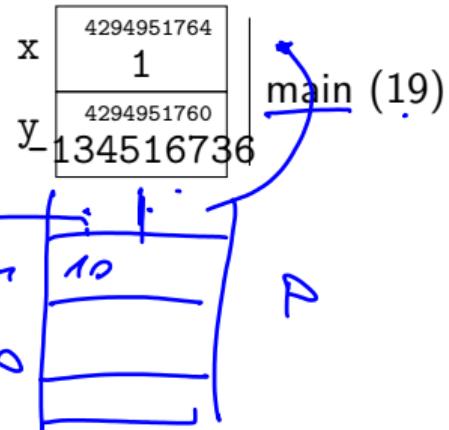
E il valore di ritorno? Due possibilità:

- in un registro: più veloce e preferibile
- se però è troppo grande (come in casi che vedremo) in un'area del record di attivazione



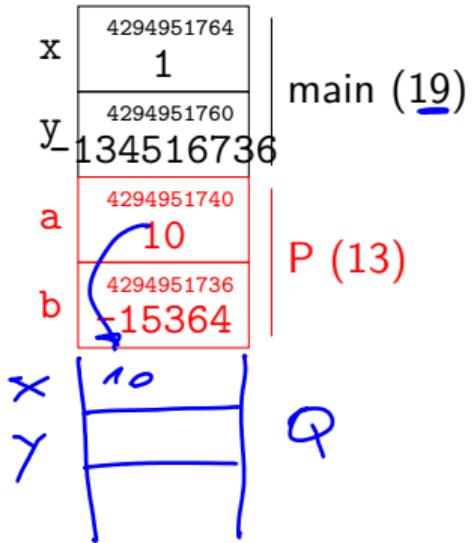
Record di attivazione

```
1 int R(int A) {  
2     int x = 2, y;  
3     y = x + A;  
4     return y;  
5 }  
6 int Q(int x) {  
7     int y;  
8     y = 2 * R(x);  
9     return y;  
10}  
11 int P(void) {  
12     int a = 10, b;  
13     b = Q(a);  
14     return b;  
15 }  
16 main() {  
17     int x = 1;  
18     int y; y = x + P();  
19 }  
20 }
```



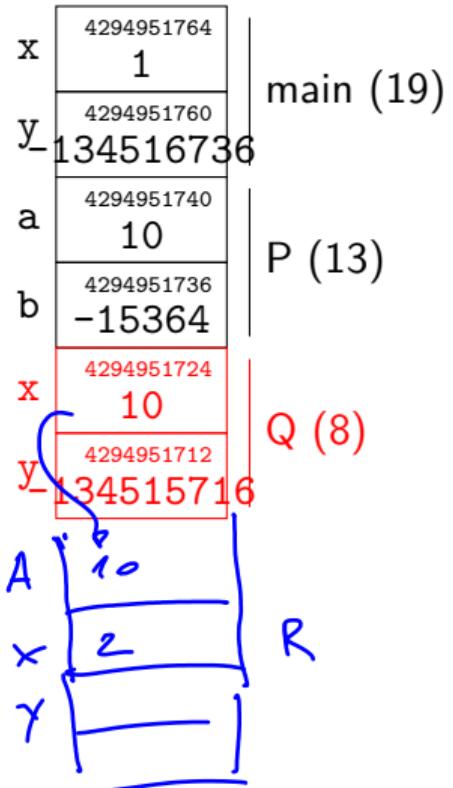
Record di attivazione

```
1 int R(int A) {  
2     int x = 2, y;  
3     y = x + A;  
4     return y;  
5 }  
6 int Q(int x) {  
7     int y;  
8     y = 2 * R(x);  
9     return y;  
10}  
11 int P(void) {  
12     int a = 10, b;  
13     b = Q(a);  
14     return b;  
15 }  
16 main() {  
17     int x = 1;  
18     int y;  
19     y = x + P();  
20 }
```



Record di attivazione

```
1 int R(int A) {  
2     int x = 2, y;  
3     y = x + A;  
4     return y;  
5 }  
6 int Q(int x) {  
7     int y;  
8     y = 2 * R(x)  
9     return y;  
10 }  
11 int P(void) {  
12     int a = 10, b;  
13     b = Q(a);  
14     return b;  
15 }  
16 main() {  
17     int x = 1;  
18     int y;  
19     y = x + P();  
20 }
```



Record di attivazione

```
1 int R(int A) {  
2     int x = 2, y;  
3     y = x + A;  
4     return y;  
5 }  
6 int Q(int x) {  
7     int y;  
8     y = 2 * R(x);  
9     return y;  
10 }  
11 int P(void) {  
12     int a = 10, b;  
13     b = Q(a);  
14     return b;  
15 }  
16 main() {  
17     int x = 1;  
18     int y;  
19     y = x + P();  
20 }
```

x	4294951764	main (19)
	1	
y	4294951760	
	134516736	
a	4294951740	P (13)
	10	
b	4294951736	
	-15364	
x	4294951724	
	10	Q (8)
y	4294951712	
	134515716	
A	4294951696	R (3)
	10	
x	4294951684	
	2	
y	4294951680	
	012	

Record di attivazione

```
1 int R(int A) {  
2     int x = 2, y;  
3     y = x + A;  
4     return y; 12  
5 }  
6 int Q(int x) {  
7     int y;  
8     y = 2 * R(x); 12  
9     return y;  
10 }  
11 int P(void) {  
12     int a = 10, b;  
13     b = Q(a);  
14     return b;  
15 }  
16 main() {  
17     int x = 1;  
18     int y;  
19     y = x + P();  
20 }
```

x	4294951764	main (19)
	1	
y	4294951760	
	134516736	
a	4294951740	P (13)
b	-15364	
x	4294951736	
	10	
y	4294951724	
	134515716	Q (8)
A	4294951696	
x	4294951684	R (4)
	2	
y	4294951680	
	12	

Record di attivazione

```
1 int R(int A) {  
2     int x = 2, y;  
3     y = x + A;  
4     return y;  
5 }  
6 int Q(int x) {  
7     int y;  
8     y = 2 * R(x);  
9     return y; 24  
10 }  
11 int P(void) {  
12     int a = 10, b;  
13     b = Q(a);  
14     return b;  
15 }  
16 main() {  
17     int x = 1;  
18     int y;  
19     y = x + P();  
20 }
```

x	4294951764	main (19)
	1	
y	4294951760	
	134516736	
a	4294951740	
b	10	P (13)
	-15364	
x	4294951736	
	24	
x	4294951724	
	10	
y	4294951712	
	24	Q (9)

Record di attivazione

```
1 int R(int A) {  
2     int x = 2, y;  
3     y = x + A;  
4     return y;  
5 }  
6 int Q(int x) {  
7     int y;  
8     y = 2 * R(x);  
9     return y;  
10}  
11 int P(void) {  
12     int a = 10, b;  
13     b = Q(a);  
14     return b; 24  
15 }  
16 main() {  
17     int x = 1;  
18     int y;  
19     y = x + P();  
20 }
```

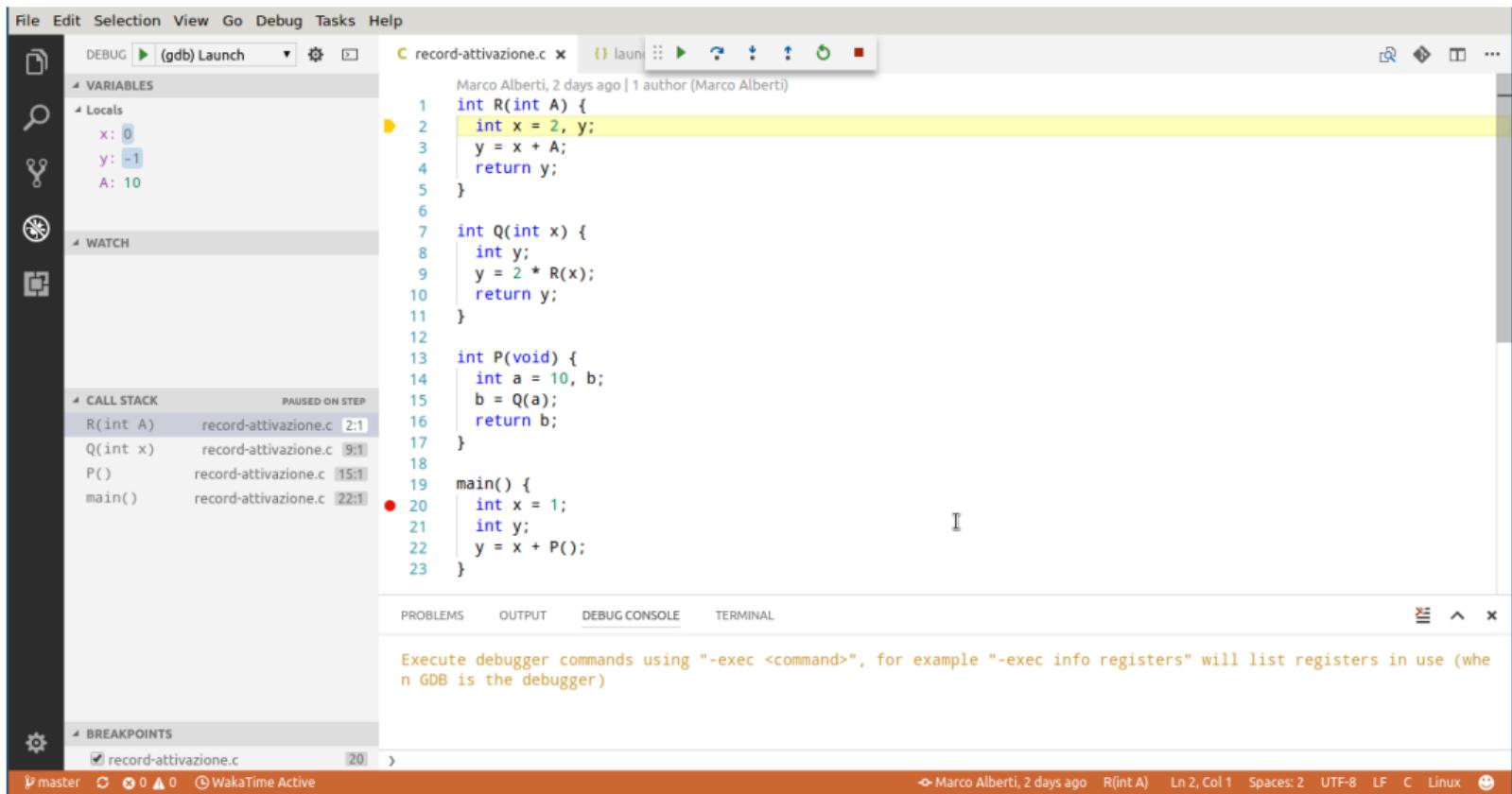
x	4294951764	main (19)
	1	
y	4294951760	
	134516736 <i>25</i>	
a	4294951740	
	10	
b	4294951736	P (14)
	24	

Record di attivazione

```
1 int R(int A) {  
2     int x = 2, y;  
3     y = x + A;  
4     return y;  
5 }  
6 int Q(int x) {  
7     int y;  
8     y = 2 * R(x);  
9     return y;  
10 }  
11 int P(void) {  
12     int a = 10, b;  
13     b = Q(a);  
14     return b;  
15 }  
16 main() {  
17     int x = 1;  
18     int y;  
19     y = x + P();  
20 }
```

x	4294951764	main (20)
	1	
y	4294951760	main (20)
	25	

Record di attivazione nel debugger



The screenshot shows the VS Code interface with the "DEBUG" tab selected. The "CALL STACK" panel is open, showing the current call stack:

- R(int A) at record-attivazione.c:2:1
- Q(int x) at record-attivazione.c:9:1
- P() at record-attivazione.c:15:1
- main() at record-attivazione.c:22:1

The source code for `record-attivazione.c` is displayed in the main editor area:

```
Marco Alberti, 2 days ago | 1 author (Marco Alberti)
1 int R(int A) {
2     int x = 2, y;
3     y = x + A;
4     return y;
5 }
6
7 int Q(int x) {
8     int y;
9     y = 2 * R(x);
10    return y;
11 }
12
13 int P(void) {
14     int a = 10, b;
15     b = Q(a);
16     return b;
17 }
18
19 main() {
20     int x = 1;
21     int y;
22     y = x + P();
23 }
```

The line `y = x + P();` is highlighted in yellow, indicating it is the current instruction being executed. The status bar at the bottom indicates the file is "Paused on Step".

Sommario

1 Funzioni

- Subroutine
- Parametri
- Valore di ritorno

2 Modello a run-time delle funzioni

3 Alcuni esercizi

4 Campo di visibilità (scope) degli identificatori

Esercizio

Valore assoluto

Scrivere una funzione di nome **valoreAssoluto** che calcoli il valore assoluto del suo parametro intero. Testarla chiamandola con parametri attuali significativi.

$$|n| = \begin{cases} n & \text{se } n \geq 0 \\ -n & \text{altrimenti} \end{cases}$$

```
int ValoreAssoluto(int n) {
```

```
    return ↗;
```

```
}
```

```
int main() {
```

```
    ↗ = ValoreAssoluto(-5)
```

```
}
```

Esercizio

Radice quadrata

Scrivere una funzione che calcoli un'approssimazione della radice quadrata di un numero reale a con il cosiddetto metodo babilonese: una successione x di approssimazioni in cui il primo elemento x_1 è 1.0 e il successore x_{k+1} di x_k è la media aritmetica fra x_k e a/x_k

Testarla chiamandola con parametri attuali significativi.

```
float radq (float a) {  
    ...  
    return ...;  
}
```

Esercizio

$$n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdots (n-1) \cdot n$$

reduce (*, 1, 1..n)

Fattoriale

Scrivere una funzione che calcoli il fattoriale del suo argomento intero. Si ricorda che il fattoriale di un numero n è il prodotto di tutti i numeri naturali compresi fra 1 e n ; quindi il fattoriale si può calcolare con un'iterazione in cui il contatore va da 1 a n e l'accumulatore, inizialmente pari a 1, è moltiplicato via via per il contatore.

Testare la funzione chiamandola con parametri attuali significativi.

```
int fatt(int n){  
    int acc = 1; i;  
    for (i=1, i<=n; i++)  
        acc = acc * i;  
    return acc;  
}
```

INPUT	OUTPUT
1	1
2	2
3	6
5	120
6	720

Esercizio

$$\max(a, b) = \begin{cases} a & a \geq b \\ b & \text{altrimenti} \end{cases}$$



Massimo

Scrivere una funzione **massimo** che restituisca il massimo dei suoi due argomenti.

```
float massimo(float a, float b){  
    if (a >= b)  
        return a;  
    else  
        return b;  
}
```

Massimo di tre

Scrivere una funzione che restituisca il massimo dei suoi tre argomenti, usando la funzione **massimo** definita al punto precedente.

```
float max3(float a, float b, float c){  
    return massimo(massimo(a, b), c);  
}
```

Esercizio

refactoring

Ristrutturare il programma `domani.c`, riportato e linkato alla slide 43, usando una funzione `bisestile` che calcoli se l'anno che riceve come argomento è bisestile, e una funzione `giorni_del_mese` che riceva come argomenti mese e anno calcoli il numero di giorni del mese nell'anno.

Aggiungere inoltre una funzione `valida` che verifica se una data è valida (ad esempio 30/2/2018 non è una data valida), e usarla per stampare un messaggio di errore e uscire con codice `-1` se l'utente scrive una data non valida.

int bisestile(int anno)

int giorni_dd_mese (int mese, int anno)

int valida (int g, int m, int a)

070_funzioni/domani.c

```

1 #include <stdio.h>
2
3 int main(void) {
4     int G, M, A;
5     int giorni_del_mese;
6
7     scanf("%d%d%d", &G, &M, &A);
8
9     if (M == 4 || M == 6 || M == 9 || M == 11)
10        giorni_del_mese = 30;
11    else if (M == 2)
12        if (A % 400 == 0 || (A % 4 == 0 && A % 100 != 0))
13            giorni_del_mese = 29;
14        else
15            giorni_del_mese = 28;
16    else
17        giorni_del_mese = 31;
18
19    if (G == 31 && M == 12)
20        printf("%d %d %d\n", 1, 1, A + 1);
21    else if (G == giorni_del_mese) // ultimo giorno del mese
22        printf("%d %d %d\n", 1, M + 1, A);
23    else
24        printf("%d %d %d\n", G + 1, M, A);
25    return 0;
26 }

```

INPUT

31 10 2020

31 12 2020

OUTPUT

1 11 2020

1 1 2021

Esercizio

$$\begin{array}{l} \text{primo}(5) \rightarrow 1 \\ \text{primo}(9) \rightarrow 0 \end{array}$$

Primi

Si scriva una funzione **primo** che restituisca **1** se il suo argomento intero è primo e **0** altrimenti.

Si utilizzi la funzione in un programma che stampi tutti i numeri primi compresi fra **2** e un numero **N** richiesto all'utente.

int primo(int n){

}

(NPVT)
N

10

OUTPVT

2

3

5

7

filter(primo, 2 . 10)
for(i=2; i <= N; i++)
if (primo(i))
printf('%d\n', i);

Sommario

1 Funzioni

- Subroutine
- Parametri
- Valore di ritorno

2 Modello a run-time delle funzioni

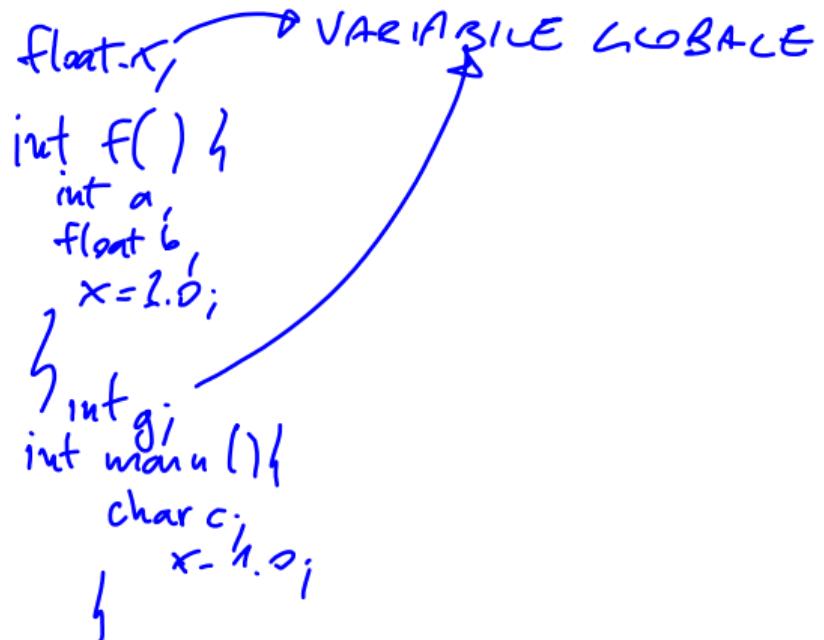
3 Alcuni esercizi

4 Campo di visibilità (scope) degli identificatori

Variabili globali

Le variabili definite nel blocco di una funzione sono dette locali.

E' possibile definire variabili fuori dalle funzioni; queste sono globali e sono visibili a tutte le funzioni definite dopo esse.



```
float x;  
int f() {  
    int a;  
    float b;  
    x=1.0;  
}  
int main() {  
    char c;  
    x=1.0;  
}
```

Esempio

Che cosa stampa questo programma?

070_funzioni/variabile-globale.c

```
1 #include <stdio.h>
2
3 int a;
4
5 void incrementa() {
6     a++;
7 }
8
9 int main() {
10    a = 1;
11    printf("%d\n", a);      1
12    incrementa();
13    printf("%d\n", a);      2
14    return 0;
15 }
```

a 12

Variabili e blocchi

In una funzione (che sia `main` o un'altra) non è necessario definire tutte le variabili all'inizio del blocco.

Le variabili possono essere definite all'inizio di qualsiasi blocco (ad esempio nel ramo falso di un `if-else`, nel corpo di un `for`, etc.).

070_funzioni/scope-for.c

```

1 #include <stdio.h>
2
3 int main(void) {
4     char risposta;
5     printf("Stampare?\n");
6     scanf("%c", &risposta);
7     if (risposta == 's') {
8         int i;
9         for (i = 0; i < 10; i++)
10            printf("*");
11            printf("\n");
12    }
13    return 0;
14 }
```

int f () {
 DEFINIZIONI

SCOPE DI risposta

SCOPE DI i

risposta=0;

Scope delle variabili

Lo **scope** (o **campo di visibilità**) di un identificatore è la parte del programma in cui l'identificatore può essere utilizzato.

Lo scope di una variabile locale è il blocco in cui è definita.

Lo scope di una variabile globale è la parte del programma che segue la sua definizione.

- Nel programma alla slide 47, quali sono gli scope delle variabili **risposta** e **i**?
- Il seguente programma è corretto? Perché?

070_funzioni/scope-blocco.c

```
1 #include <stdio.h>
2
3 int main(void) {
4     {int i = 1;} SCOPE DI:
5     printf("%d\n", i);
6     return 0;
7 }
```

Identifieri uguali

E' vietato definire variabili con lo stesso nome nello stesso scope.

070_funzioni/stesso-nome-stesso-scope.c

```
1 #include <stdio.h>
2
3 int main(void) {
4     int i;
5     int i = 1; ERRORE
6     printf("%d\n", i);
7     return 0;
8 }
```

Identifieri uguali

E' però consentito definire variabili con lo stesso nome in scope diversi, anche uno contenuto nell'altro.

070_funzioni/stesso-nome-scope-diversi.c

```
1 #include <stdio.h>
2
3 int i = 0;
4
5 int main(void) {
6     int i = 1; s2
7     {
8         int i = 2; s1
9         printf("%d\n", i);
10    }
11    printf("%d\n", i); s2
12    return 0;
13 }
```

Ma qual è la **i** alla riga 9? E quella alla riga 11?

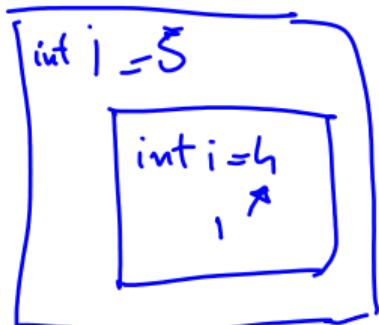
Lo scope interno maschera quello esterno

A partire dall'espressione variabile, è possibile costruire una sequenza di scope numerati come segue:

- 1 lo scope del blocco in cui si trova l'espressione
- 2 lo scope del blocco che contiene quello al punto 1
- n ... e così via fino allo scope del corpo della funzione
- n+1 infine, lo scope globale

riga 10
s1 i=2
riga 11
s2 i=1

L'identificatore si riferisce alla variabile con lo scope con numero minimo (cioè quello più interno).



Esercizio

Che cosa stampa questo programma?

070_funzioni/scope-domanda1.c

```
1 #include <stdio.h>
2
3 int a = 1;
4
5 int main() {
6     while (a--) {
7         {
8             int a = 3;
9             printf("%d\n", a);
10        }
11        printf("%d\n", a);
12    }
13 }
```

a 10 - 1

a 3

OUTPUT

3

0

Esercizio

Il seguente programma è valido? Perché?

070_funzioni/scope-domanda2.c

```
1 #include <stdio.h>
2
3 int main() {
4     while (a--) {
5         {
6             int a = 3;
7             printf("%d\n", a);
8         }
9         printf("%d\n", a);
10    }
11 }
12
13 int a = 1; SCOPE
```

Osservazione

Secondo le regole di scope, nessuna funzione ha mai accesso alle variabili locali di un'altra funzione (almeno, attraverso il suo identificatore). Quindi variabili locali di funzioni diverse, anche se hanno lo stesso nome, sono sempre variabili diverse.

Questo è un altro vantaggio delle funzioni rispetto al pattern sottoprocedura:

- le variabili sono definite dove vengono usate
- è possibile riutilizzare i nomi

