



Sottoprogrammi

Antonella Santone



Cominciamo con un esempio

... dati due interi positivi calcolare il massimo dei rispettivi fattoriali ...

```
input n, m
if (n >= 0 && m >= 0) {
    Calcola il fattoriale di n e assegnalo ad fn
    Calcola il fattoriale di m e assegnalo ad fm
    Calcola il massimo di fn e fm e assegnalo a max
    output max }
else output messaggio di errore
```



Con le vostre conoscenze

```
#include <stdio.h>

main(){
    int n, m, fn, fm, max, i;
    scanf("%d %d", &n, &m);
    if (n >= 0 && m >= 0){
        fn = 1;
        for (i=1; i<=n; i++)
            fn = fn*i;
        fm = 1;
        for (i=1; i<=m; i++)
            fm = fm*i;
        if (fn >= fm) max=fn;
        else max=fm;
        printf("Il massimo = %d", max);
    }
    else printf("errore");
}
```



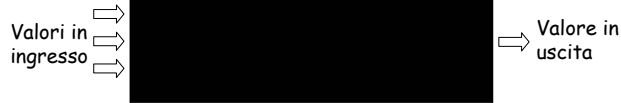
Concetto di sottoprogramma

E' possibile aggregare gruppi di istruzioni per formare "sottoprogrammi"

I sottoprogrammi si usano per evitare di replicare porzioni di codice sorgente

- invocazione sottoprogramma = esecuzione della porzione di codice corrispondente
- Tante chiamate, una sola porzione di codice

Sottoprogramma



Scatola nera:
a determinati valori in ingresso fa corrispondere un valore in uscita

Utilizzando un sottoprogramma

```
#include <stdio.h>

main(){
    int n, m, fn, fm, max, i;
    scanf("%d %d", &n, &m);
    if (n >= 0 && m >= 0){
        fn = 1;
        for (i=1; i<=n; i++)
            fn = fn*i;
        fm = 1;
        for (i=1; i<=m; i++)
            fm = fm*i;
        if (fn >= fm) max=fn;
        else max=fm;
        printf("Il massimo = %d", max);
    }
    else printf("errore");
}
```

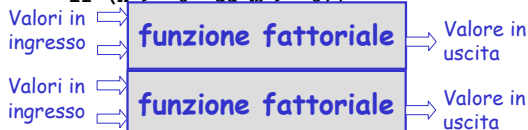
Utilizzando un sottoprogramma

```
#include <stdio.h>

main(){
    int n, m, fn, fm, max, i;
    scanf("%d %d", &n, &m);
    if (n >= 0 && m >= 0){
        if (fn >= fm) max=fn;
        else max=fm;
        printf("Il massimo = %d", max);
    }
    else printf("errore");
}
```

funzione fattoriale

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



Perché usare sottoprogrammi

- Evitare ripetizione
- Facile correggere
- Facile modificare
- Permettere di suddividere il problema in sottoproblemi
→ migliorare la struttura del programma
- Aumentare la leggibilità del programma
- Permettere a più persone di lavorare allo stesso progetto

Divide et Impera

I sottoprogrammi sono utilizzati per semplificare la soluzione di problemi complessi attraverso il noto principio del **divide et impera**:

- dividi il problema in sottoproblemi
- risolvi separatamente i sottoproblemi
- ricombina i risultati

Torniamo all'esempio

... dati due interi positivi calcolare il massimo dei rispettivi fattoriali ...

```
input n, m
if (n >= 0 && m >= 0) {
    Calcola il fattoriale di n e assegnalo ad fn
    Calcola il fattoriale di m e assegnalo ad fm
    Calcola il massimo di fn e fm e assegnalo a max
    output max }
else output messaggio di errore
```

NB: ogni frase in corsivo costituisce un sottoproblema che può essere risolto con un sottoprogramma

Comunicazione tra sottoprogrammi

Definire le modalità di passaggio delle informazioni (dati di ingresso) necessarie affinché un sottoprogramma possa risolvere il corrispondente sottoproblema in modo autonomo e inviare i risultati ottenuti (dati di uscita) al sottoprogramma che lo utilizza

Esempio:

al sottoprogramma che risolve il sottoproblema *calcola il fattoriale di n ed assegnalo ad fn* bisogna passare il valore n; il sottoprogramma restituisce il valore da assegnare ad fn

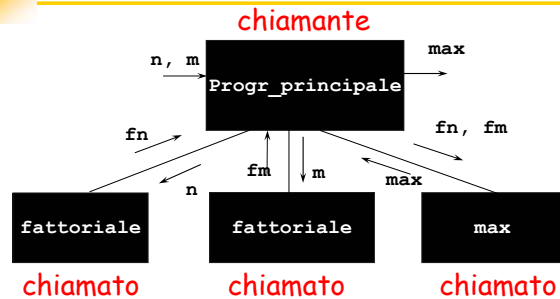
Comunicazione tra sottoprogrammi (cont.)

Il meccanismo che consente ad un sottoprogramma di utilizzare un altro sottoprogramma come se fosse una operazione elementare è il meccanismo di **chiamata**

Il controllo dell'esecuzione viene trasferito dal sottoprogramma **chiamante** al sottoprogramma **chiamato**, insieme ai dati di ingresso (**attivazione**)

Al termine dell'esecuzione, il sottoprogramma chiamato restituisce il controllo al sottoprogramma chiamante insieme ai dati di uscita

Decomposizione funzionale: esempio



Struttura dei sottoprogrammi

Un sottoprogramma è composto da:

- ❖ **intestazione** (interfaccia), che specifica il nome del sottoprogramma
- ❖ **lista dei parametri** (nome, tipo) scambiati
- ❖ **un corpo**, che definisce le azioni e i dati (locali) utilizzati dal sottoprogramma per risolvere il sottoproblema

La visione di chi utilizza un sottoprogramma è diversa dalla visione di chi lo progetta e implementa:

chi utilizza un sottoprogramma è interessato a come è definita la sua interfaccia, non a come è realizzato

Definizione di funzione

L'intestazione di una funzione specifica:

- il tipo di ritorno,
- il nome;
- lista degli argomenti (parametri formali) di una funzione per ogni argomento viene indicato il tipo e il nome (gli argomenti sono separati da virgole)

return: istruzione che indica il valore (espressione) restituito dalla funzione

```
return (espressione) ;
```

Sintassi

```
tipo_restituito nome_funzione (par1, par2, ...) intestazione
{
  dichiarazione variabili locali corpo
  istruzioni
}
```

par1, par2, ... sono la definizione degli argomenti da passare alla funzione all'atto della chiamata (ad es. `int i`). Una funzione può anche essere senza parametri, in tal caso l'elenco dei parametri sarà vuoto. Tuttavia, anche se non vi è alcun parametro, è richiesta la presenza delle parentesi

tipo_restituito è il tipo del dato che viene restituito come risultato dalla funzione. Se manca la definizione del tipo di dato restituito dalla funzione **tipo_restituito**, il C assume che il risultato della funzione è di tipo `int`

nome_funzione valgono le regole in uso per gli identificatori

Ancora sulla sintassi ...

Sappiamo che nella dichiarazione di variabili, è possibile dichiarare più variabili di un tipo comune utilizzando un elenco di nomi di variabili separati da virgole:

```
int a , b ;
```

Al contrario, tutti i parametri della funzione devono essere dichiarati singolarmente e per ognuno si deve specificare sia il tipo che il nome:

```
int somma (int a, int b)
```

return

La funzione restituisce il risultato mediante un'istruzione detta `return()`, che ha la forma:

```
return espressione;  
o  
return (espressione);
```

La `return` termina la funzione e restituisce il controllo al chiamante.

Se la funzione non deve restituire nessun valore, si dichiara il `tipo_restituito` come `void`, e non si esegue la `return` o la si esegue senza passare alcun argomento (es.: `return;`).

Esempi di definizioni di funzioni

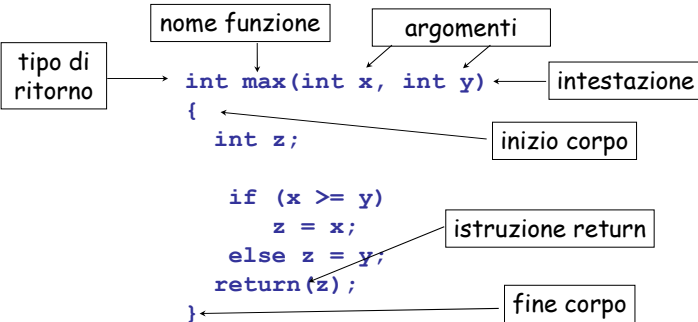
```
void nothing() {}
```

```
int twice(int x)  
{  
    return (2*x);  
}
```

Sottoprogrammi in C

In C esiste un'unica primitiva per implementare sottoprogrammi: la **funzione**; non esiste la distinzione che esiste in Pascal tra funzioni e procedure, in C sono tutte funzioni, che possono restituire un qualche risultato oppure no, nel qual caso restituiscono `void`

Esempio



Errori tipici

- Inserire un punto e virgola dopo la parentesi destra che chiude l'elenco dei parametri in una definizione di funzione
- Restituire un dato da una funzione il cui valore di ritorno sia stato dichiarato di tipo `void`
- Dimenticare di restituire un valore da una funzione che dovrebbe farlo

Chiamata delle funzioni

Le chiamate ad ogni funzione in C si effettuano chiamando il nome della funzione seguita dalle parentesi tonde, aperta e chiusa, all'interno delle quali vengono passati i parametri necessari

Esempio: `max(a, b);`

Anche se la funzione non richiede nessun argomento, nella chiamata il suo nome deve essere seguito dalle parentesi tonde aperte e chiuse

Esempio: `nothing();`

Il `main` stesso è una funzione

Parametri formali e parametri attuali

parametri formali

parametri dichiarati nell'intestazione di un sottoprogramma

parametri attuali

parametri che compaiono in una chiamata ad un sottoprogramma

All'atto della chiamata i parametri formali del sottoprogramma vengono sostituiti dai parametri attuali (attualizzazione)

La corrispondenza tra parametri attuali e parametri formali deve essere in numero, ordine e tipo

Esempio completo

```
int max(int x, int y)
{
    int z;
    if (x >= y)
        z = x;
    else z = y;
    return (z);
}
main()
{
    int a,b;
    scanf("%d %d", &a, &b);
    printf("Massimo = %d", max(a, b));
}
```

parametri formali

istruzione di chiamata e
parametri attuali

Al termine della chiamata, verrà
stampato il massimo tra i valori
di a e b

Chiamata di una funzione

Una funzione può essere chiamata specificando
il nome e la lista di parametri attuali

```
main()
{
    int a,b;

    scanf("%d %d", &a, &b);
    printf("Massimo = %d", max(a, b));
}
```

chiamata

parametri attuali

Definizione e dichiarazione di funzione

Definizione: intestazione + corpo

```
tipo_ritorno Nome (lista di parametri)
{
    dichiarazione di variabili locali
    istruzioni
}
```

Dichiarazione: prototipo

```
tipo_ritorno Nome (lista di parametri);
```

Per poter usare un identificatore occorre dichiararlo!!!

Dichiarazione di funzione - prototipo

Per poter essere chiamata, una funzione deve essere stata
prima dichiarata

il compilatore ha bisogno di controllare il nome e la
corrispondenza di parametri attuali e parametri formali

In realtà, quello che serve al compilatore è solo l'intestazione
della funzione ... non il corpo.

Ciò che conta in una dichiarazione è il tipo (la lista dei tipi dei
parametri formali)

il prototipo di una funzione è costituito dalla sua sola
intestazione terminata da un punto e virgola

Esempio: int max (int x, int y);

Prototipo

Il prototipo di una funzione rappresenta una dichiarazione di funzione (indicandone la sola intestazione) antecedente alla sua definizione.

Esempi

```
int function(int x, char c);  
int function(int, char);
```

L'uso dei prototipi è obbligatorio quando la definizione di una funzione avviene successivamente al suo utilizzo.

... in sintesi

Il prototipo o dichiarazione della funzione è un'istruzione che ripete l'intestazione della funzione, senza il codice.

Esempio

```
double somma(double x, double y) ;           dichiarazione  
  
main() {  
    double A=10, B=29, C;  
    C = somma(A,B);                           chiamata  
}  
  
double somma(double x, double y) {          definizione  
    return (x+y) ;  
}
```

Perché usare il prototipo?

- dare visibilità alla funzione quando il luogo in cui è chiamata precede il luogo in cui è definita (se stiamo sullo stesso file), altrimenti il compilatore non sa cos'è il simbolo nome della funzione.
- informare il compilatore su come deve trattare il dato restituito dalla funzione, e su come passare i dati agli argomenti della funzione. Quindi il prototipo e la definizione della funzione devono essere coerenti

Un esempio completo

```
#include <stdio.h>  
int max(int x, int y); ← dichiarazione  
main()  
{  
    int a,b;  
    scanf("%d %d", &a, &b);  
    printf("Massimo = %d", max(a, b));  
}  
int max(int x, int y)  
{  
    int z;  
    if (x >= y) ← definizione  
        z = x;  
    else z = y;  
    return (z);  
}
```

Errore tipico

Dimenticare il punto e virgola alla fine del prototipo di funzione

Stile

Programma in un unico file

I alternativa

```
#include <stdio.h>
...
prototipo f1;
prototipo f2;
...
main() {      }
definizione f1
definizione f2
```

II alternativa

La definizione può servire anche come prototipo → rimuovere dall'elenco i prototipi di funzione e scrivere la definizione di ogni funzione

```
#include <stdio.h>
...
definizione f1
definizione f2
...
main() {      }
```

Torniamo all'esempio

... dati due interi positivi calcolare il massimo dei rispettivi fattoriali ...

```
input n, m
if (n >= 0 && m >= 0) {
    Calcola il fattoriale di n e assegno ad fn
    Calcola il fattoriale di m e assegno ad fm
    Calcola il massimo di fn e fm e assegno a max
    output max }
else output messaggio di errore
```

NB: ogni frase in corsivo costituisce un sottoproblema che può essere risolto con un sottoprogramma

.. diventa stile 1

```
#include <stdio.h>
int fattoriale(int n);
int max(int x, int y);
main() {
    int n, m, fn, fm;
    scanf("%d %d", &n, &m);
    if (n >= 0 && m >= 0) {
        fn = fattoriale(n);
        fm = fattoriale(m);
        printf("Il massimo = %d", max(fn, fm));
    }
    else printf("errore");
}
int fattoriale(int n) {
    int fatt, i;
    fatt = 1;
    for (i=1; i<=n; i++)
        fatt = fatt*i;
    return (fatt);
}
int max(int x, int y) {
    int z;
    if (x >= y)
        z = x;
    else z = y;
    return (z);
}
```

.. diventa stile 2

```
#include <stdio.h>

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

int max(int x, int y){
    int z;
    if (x >= y)
        z = x;
    else z = y;
    return (z);
}

main(){
    int n, m, fn, fm;
    scanf("%d %d", &n, &m);
    if (n >= 0 && m >= 0){
        fn = fattoriale(n);
        fm = fattoriale(m);
        printf("Il massimo = %d", max(fn, fm));
    }
    else printf("errore");
}
```

Esempi di funzioni

Funzione che somma due valori di tipo int e restituisce un int:

```
int somma(int a, int b) {
    int sum;
    sum = a+b;
    return (sum);
}
```

La chiamata della funzione viene fatta così:

```
main() {
    int A=23;
    int B=-31;
    int risultato;
    risultato = somma(A,B);
    printf("somma= %d\n", risultato);
}
```

Esempi di funzioni

Funzione che non restituisce alcun valore:

```
void somma(int a, int b) {
    int sum;
    sum = a+b;
    printf("somma= %d\n", sum); /* non serve la return */
}
```

La chiamata della funzione viene fatta così:

```
main() {
    int A = 23;
    int B = -31;
    somma(A,B);
}
```

Cosa succede ...

Se compiliamo il seguente programma?

**conflicting types
for 'cubo'**

```
#include <stdio.h>

double cubo(float x);
float cubo(int x);
main()
{
    int a;
    scanf("%d", &a);
    printf("Cubo = %d", cubo(a));
}

double cubo(float x)
{
    return (x*x*x);
}

float cubo(int x)
{
    return (x*x*x);
}
```

In C non è ammesso che più funzioni abbiano lo stesso nome e parametri e valori di ritorno diversi

Passaggio degli argomenti

Parametri formali/attuali

I **parametri formali** sono quelli dichiarati per tipo, numero e ordine nella definizione della funzione

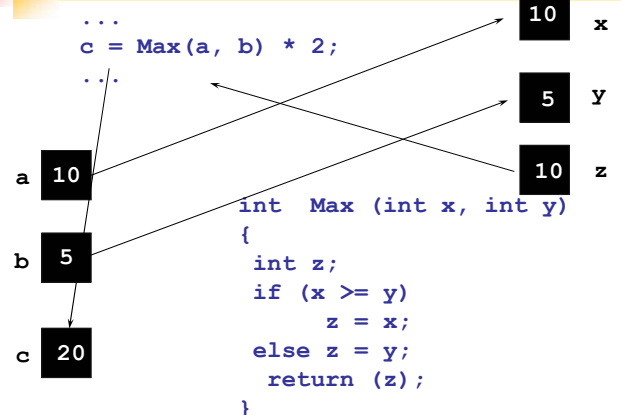
I **parametri attuali** sono invece quelli passati alla funzione all'atto della chiamata

Passaggio di parametri per valore

In C il passaggio dei parametri avviene sempre e soltanto per **VALORE**. All'atto della chiamata, i valori dei parametri attuali (di input) sono copiati nelle locazioni di memoria dei corrispondenti parametri formali (i parametri attuali possono essere delle espressioni)

Eventuali modifiche dei parametri formali non si estendono ai parametri attuali (se questi ultimi sono delle variabili)

Passaggio di parametri per valore



Corrispondenza tipo

In C deve esistere una coerenza di tipo e di numero tra parametri formali e parametri attuali.

Corrispondenza non perfetta: possono essere effettuate delle conversioni implicite

Corrispondenza tipo: esempio

```
#include <stdio.h>

double cubo(float x);
main()
{
    double c;
    int a;
    scanf("%d", &a);
    c = cubo(a);
    printf("Cubo = %lf", c);
}

double cubo(float x)
{
    return (x*x*x);
}
```

Valido in C perché la variabile `a` di tipo `int` viene convertita in `double`

Nota sul passaggio per valore

```
#include <stdio.h>
void add3 (int n) ;
```

Funzioni che non restituiscono alcun valore

```
main()
{
    int n = 5;
    add3 (n);
    printf("Il valore di n nel progr. princ. = %d", n);
}
```

2. il valore di n è 5

```
void add3(int n)
{
    n = n + 3 ;
    printf("Il valore di n nella funzione = %d", n);
}
```

1. il valore di n è 8

Nota sul passaggio per valore (cont.)

I dati di tipo semplice (`char`, `int`, ecc.), che vengono passati come argomenti ad una funzione, vengono passati per valore:

al momento della esecuzione della funzione, il valore degli argomenti viene copiato sullo stack, e la funzione usa (ed eventualmente modifica) questa copia degli argomenti

Conseguenza

dato originale rimane invariato dopo che la funzione ha restituito il controllo al chiamante

Se si vuole che un argomento di tipo semplice passato alla funzione conservi le modifiche apportate dalla funzione durante l'esecuzione della funzione, l'argomento deve essere passato per **indirizzo**, cioè alla funzione deve essere passato l'indirizzo della variabile, in modo che la funzione (tramite l'indirizzo) modifica la variabile originale

Passaggio degli argomenti: array

I dati di tipo **array** che vengono passati come argomenti ad una funzione, in C vengono passati per indirizzo, nel senso che passando l'array per nome si passa l'indirizzo in cui comincia l'array.

Ciò significa che quando, all'atto della chiamata, passiamo ad una funzione il nome di un vettore, passiamo l'indirizzo del primo elemento del vettore, e non tutti i dati del vettore (100 interi dell'esempio)

```
main() {
    int vet[100];
    modifica_vet (vet , 100 );
}
```

Esempio completo

```
#include <stdio.h>
void modifica_vet(int vet[]);

main() {
    int vet[10], i;
    for (i=0; i<10; i++){
        printf("dammi l'el. %d-esimo di a\n", i);
        scanf("%d", &vet[i]);
    }
    modifica_vet (vet);
    printf ("%d\n", vet[0]);
}

void modifica_vet(int vet[])
{
    vet[0]=0; /* modifica conservata fuori dalla funzione */
}
```

Altri tipi di passaggi

Molti linguaggi mettono a disposizione il passaggio per riferimento (by reference)

non si trasferisce una copia del valore del parametro attuale, ma si trasferisce un riferimento al parametro, in modo da dare alla funzione accesso diretto al parametro in possesso del chiamante;

la funzione accede e modifica direttamente il dato del chiamante

Il C non supporta direttamente il passaggio per riferimento, lo fornisce indirettamente solo per alcuni tipi di dati, quindi, occorre costruirselo quando serve.

C++ e Java invece lo forniscono

Come realizzare in C il passaggio per riferimento

E' possibile realizzare l'effetto di un passaggio per indirizzo nel seguente modo:

- utilizzando il tipo puntatore per la definizione dei parametri formali;
- usando l'operatore di dereferenziazione di puntatore all'interno del corpo del programma (operatore *);
- passando al momento della chiamata della funzione come parametro attuale un indirizzo di variabile (operatore &)

Scambio dei parametri

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

Qual è l'output del programma ?
3 5

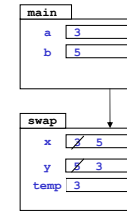
Ciò accade perché la funzione `swap` agisce solo su una copia delle variabili `a` e `b`

```
main() {
    int a = 3;
    int b = 5;
    swap(a,b);
    printf("%d %d\n",a,b);
}
```

Per far sì che la funzione agisca sulle variabili stesse e non su delle copie occorre passare gli indirizzi delle variabili

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

main() {
    int a = 3;
    int b = 5;
    swap(a,b);
    printf("%d %d\n",a,b);
}
```



Output:
3 5

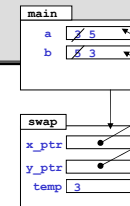
Scambio dei parametri (cont.)

```
void swap(int *x_ptr, int *y_ptr) {
    int temp;
    temp = *x_ptr;
    *x_ptr = *y_ptr;
    *y_ptr = temp;
}
```

```
main() {
    int a = 3;
    int b = 5;
    swap(&a,&b);
    printf("%d %d\n",a,b);
}
```

```
void swap(int *x_ptr, int *y_ptr) {
    int temp;
    temp = *x_ptr;
    *x_ptr = *y_ptr;
    *y_ptr = temp;
}

main() {
    int a = 3;
    int b = 5;
    swap(&a,&b);
    printf("%d %d\n",a,b);
}
```



Output:
5 3

Ordine di valutazione degli argomenti passati alle funzioni

Il compilatore C opera in modo che le espressioni passate come argomenti alle funzioni sono prima valutate, ed il risultato della valutazione viene scritto sullo stack per essere disponibile al codice che implementa la funzione stessa.

Sorgono due problemi:

- in che ordine vengono valutate le espressioni?
- in che ordine vengono scritti sullo stack i risultati delle valutazioni?

La risposta alle due domande è la stessa:

vengono prima valutate (e il risultato scritto sullo stack) le espressioni passate come ultimo argomento della funzione (le più a destra), e poi via via quelle più a sinistra

Esempio

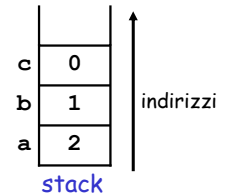
```
#include <stdio.h>

void stampa3( int a, int b, int c){
printf("a=%d b=%d c=%d\n", a, b, c);
}

main() {
int x=0;
stampa3 (x++, x++, x++);
}
```

L'output del programma sarà del tipo:
a=2 b=1 c=0

(c valutato per primo)



Tipi di dato restituiti dalle funzioni

Le funzioni possono restituire:

- dati di tipo semplice, come char, int, long, float, double, puntatori a void, o puntatori a qualche tipo di dato,
- ma anche strutture (struct) o puntatori a struct.

All'atto della chiamata, il valore restituito da una funzione per esempio `double somma(double f, double g);`

- può essere utilizzato come espressione booleana:
`if (somma(a,b) > 100.3)`
- può essere utilizzato come membro di destra in un'istruzione di assegnamento,
`f = somma(a,b);`
- oppure può non essere considerato affatto
`somma(a,b);`

Tipi di dato restituiti dalle funzioni

Vediamo un esempio di restituzione di una struct.

```
struct point { int x; int y; };

struct point crea_point( int a, int b ) {
    struct point p;
    p.x = a;
    p.y = b;
    return p;
}

main() {
    struct point p1;
    int x=21, y = -10987;
    p1 = crea_point( x, y );
    printf ( "p1.x=%d p1.y=%d \n" , p1.x, p1.y );
}
```

Visibilità

Concetto di blocco

Un blocco è composto da due parti sintatticamente racchiuse tra parentesi graffe:

- una parte dichiarativa (facoltativa);
- una sequenza di istruzioni

Blocchi annidati

Visibilità

Le regole di visibilità (scope) sono le regole che determinano la visibilità delle variabili di un programma

- in quali punti del programma è lecito usare un identificatore dichiarato in un altro punto??
- se un identificatore è dichiarato in due punti diversi con significati diversi, quale significato bisogna assumere in un generico punto del programma??

Esempio

```
#include <stdio.h>
void f() {
    int i = 2;
    printf("%d\n", i);
}
main() {
    int i = 1, j = 5;
    printf("%d\n", i);
    printf("%d\n", j);
    {
        int i = 10;
        printf("%d\n", i);
        printf("%d\n", j);
    }
    printf("%d\n", i);
    f();
    printf("%d\n", i);
}
```

Posso fare `printf("%d\n", i);` qui?

Visibilità

Le variabili assumono caratteristiche diverse, in particolare caratteristiche di visibilità da parte delle funzioni, in dipendenza della posizione in cui avviene la dichiarazione.

A seconda della posizione in cui avviene la dichiarazione, si distinguono tre tipi di variabili:

- Variabili Locali
- Variabili Globali
- Parametri Formali

Variabile locale

Una variabile locale può essere dichiarata dentro un qualunque blocco, ma in questo caso sempre e solo all'inizio del blocco, cioè mai dopo che nel blocco sia stata scritta un'istruzione diversa da una dichiarazione, ed in tal caso:

- la variabile verrà detta locale al blocco,
- potrà essere acceduta solo dall'interno del blocco stesso, cioè non è visibile fuori dal blocco,
- avrà un ciclo di vita che inizierà nel momento in cui il controllo entra nel blocco, e terminerà nel momento in cui il controllo esce dal blocco

durata

Variabile globale

Le variabili globali sono quelle variabili che sono dichiarate fuori da tutte le funzioni, in una posizione qualsiasi del file.

Una tale variabile allora verrà detta globale, perché

- potrà essere acceduta da tutte le funzioni che stanno nello stesso file ma sempre dopo la dichiarazione della variabile stessa,
- avrà durata pari alla durata in esecuzione del programma

durata

Variabili globali (cont.)

L'uso eccessivo di variabili globali è in genere sconsigliato nella pratica dello sviluppo del software poiché la loro presenza facilita l'occorrere di effetti collaterali che possono portare a errori di programmazione molto difficili da individuare e correggere

Parametri formali

i parametri formali potranno essere acceduti solo dall'interno della funzione in cui sono stati dichiarati (come le variabili locali);

durata

avranno un ciclo di vita che inizierà nel momento in cui il controllo entra nella funzione stessa, e terminerà nel momento in cui il controllo esce dal blocco

I parametri formali vengono caricati sullo stack quando il controllo entra nel blocco considerato, e vengono eliminati quando il controllo esce dal blocco in cui sono state dichiarate

Problemi con le variabili globali

```
#include <stdio.h>

int K=2; /* var. globale visibile da tutte le funzioni */
double g=13;

void funzione1(void) {
    printf("g = %f \n", g); /* stampa g, cioè 13, corretto */
    printf("i = %d \n", i); /* NON VEDE i, ERRORE */
}

main() {
    int i=34;
    printf("i = %d \n", i); /* stampa i cioè 34, corretto */
    int J=0; /* ERRORE, dichiarazione dopo istr. */
    printf("K = %d \n", K); /* stampa K cioè 2, corretto */
    printf("g = %f \n", g); /* stampa g = 13.0 */
    funzione1();
}
```

Mascheramento

Nei linguaggi dotati del concetto di blocco una variabile visibile all'interno di un blocco è in generale visibile anche all'interno di eventuali blocchi annidati.

Le regole fondamentali di visibilità sono in genere modificate dalla regola speciale dello shadowing:

una variabile locale **nasconde** una eventuale variabile globale omonima : se in un programma è definita una variabile globale **var** e in un determinato sottoprogramma viene definita una variabile locale omonima, il sottoprogramma in questione perde la visibilità della variabile globale, nascosta da quella locale

Esempio

Cosa succede stesso identificatore per dichiarazioni diverse?

```
int x=6; /* globale */
f(){
    int x; /* x locale che nasconde x globale*/
    x=1; /* assegna 1 alla x locale */
    { int x; /* nasconde il primo x locale*/
      x=2; /* assegna 2 alla seconda x locale */
    }

    x=3 /* assegna 3 alla prima x locale */
}
printf("%d", x); /* stampa x globale che vale 6*/
```

Esempio completo

```
#include<stdio.h>

int x=6;          /* globale */

f(){
  int x;         /* x locale che nasconde x globale*/
  x=1;          /* assegna 1 alla x locale */
  { int x;      /* nasconde il primo x locale*/
    x=2;        /* assegna 2 alla seconda x locale */
    printf("%d\n", x);
  }
  x=3;          /* assegna 3 alla prima x locale */
  printf("%d\n", x);
}
main()
{
  printf("%d\n", x);
  f();
}
```

Esempio

```
int x;
f(){
  int y;
  y=1;
}
```

La visibilità di **y** si estende dal punto di definizione fino alla fine del blocco di appartenenza

Esempio - parametri formali

I parametri formali di una funzione hanno un campo di visibilità che si estende dall'inizio alla fine del blocco, quindi sono considerati variabili locali alla funzione

```
int x;
g(int y, char z){
  int k;
  int l;
  ...
}
```

Le variabili **y** e **z** sono locali alla funzione **g** ed hanno una visibilità che si estende alla parentesi graffa aperta a quella chiusa. **k** e **l** hanno la stessa visibilità. Per questo motivo

```
f(int x){
  int x;
}
```

è errata: definiamo due volte la variabile **x** nello stesso blocco

Cosa stampa

```
#include <stdio.h>
void f() {
  int i = 2;
  printf("%d\n", i);
}
main() {
  int i = 1, j = 5;
  printf("%d\n", i);
  printf("%d\n", j);
  {
    int i = 10;
    printf("%d\n", i);
    printf("%d\n", j);
  }
  printf("%d\n", i);
  f();
  printf("%d\n", i);
}
```

Soluzione

1
5
10
5
1
2
1

```
#include <stdio.h>
void f() {
    int i = 2;
    printf("%d\n", i);
}
main() {
    int i = 1, j = 5;
    printf("%d\n", i);
    printf("%d\n", j);
    {
        int i = 10;
        printf("%d\n", i);
        printf("%d\n", j);
    }
    printf("%d\n", i);
    f();
    printf("%d\n", i);
}
```

Sottoprogrammi

esempio

Gestire vettori

- riempimento di un vettore;
- stampa degli elementi del vettore
- ricerca del massimo elemento di un vettore

Riempimento

```
void riemp(int vet[], int lung){
    int i;
    for (i = 0; i < lung; i++){
        printf("Dammi il elemento [%d]: ", i);
        scanf("%d", &vet[i]);
    }
}
```

Stampa

```
void stampa (int vet[], int lung){
    int i;
    printf("Il vettore: ");
    for(i = 0; i < lung; i++)
        printf("%d\t", vet[i]);
}
```

Ricerca del massimo

```
int max(int vet[], int lung){
    int i, max=vet[0];
    for (i=1; i<lung; i++)
        if (vet[i] > max) max=vet[i];
    return (max);
}
```

```
#include <stdio.h>
#define N 10

void riemp(int vet[], int lung);
void stampa (int vet[], int lung);
int max(int v[], int lung);

main() {
    int vet[N];
    riemp(vet, N);
    printf("il massimo di vet = %d\n", max(vet,N));
    stampa(vet,N);
}

void riemp(int vet[], int lung){
    int i;
    for (i = 0; i < lung; i++){
        printf("Dammi il elemento [%d]: ", i);
        scanf("%d", &vet[i]);
    }
}

void stampa (int vet[], int lung){
    int i;
    printf("Il vettore: ");
    for(i = 0; i < lung; i++)
        printf("%d\t", vet[i]);
}

int max(int vet[], int lung){
    int i, max=vet[0];
    for (i=1; i<lung; i++)
        if (vet[i] > max) max=vet[i];
    return (max);
}
```

Programma completo