Funzioni, Stack e Visibilità delle Variabili in C

Laboratorio di Programmazione I

Corso di Laurea in Informatica
A.A. 2019/2020

Argomenti del Corso

Ogni lezione consta di una spiegazione assistita da slide, e seguita da esercizi in classe

- Introduzione all'ambiente Linux
- Introduzione al C
- Tipi primitivi e costrutti condizionali
- Costrutti iterativi ed array
- Funzioni, stack e visibilità variabili
- Puntatori e memoria
- Debugging
- Tipi di dati utente
- Liste concatenate e librerie
- Ricorsione

Sommario

Outline

Funzioni in C

Per definire un programma complesso è necessario dividerlo in parti separate: ***modularizzazione***.

Una funzione permette di definire una sorta di sottoprogramma con:

- parametri in ingresso;
- parametri in uscita.

La scrittura di un programma in funzioni (***astrazione funzionale***) consente di realizzare e riusare operazioni complesse definendole come funzioni da invocare quando necessario.

Funzioni in C (2)

Un programma C è definito come un insieme di funzioni (una obbligatoria: il ***main***).

Una funzione prende in ingresso un insieme di argomenti e ritorna un (*unico*) valore.

Vedremo di seguito:

- Come si definiscono le funzioni (*definizione di funzione*);
- Come si usano le funzioni (chiamata o *attivazione* di funzione).

Un primo esempio

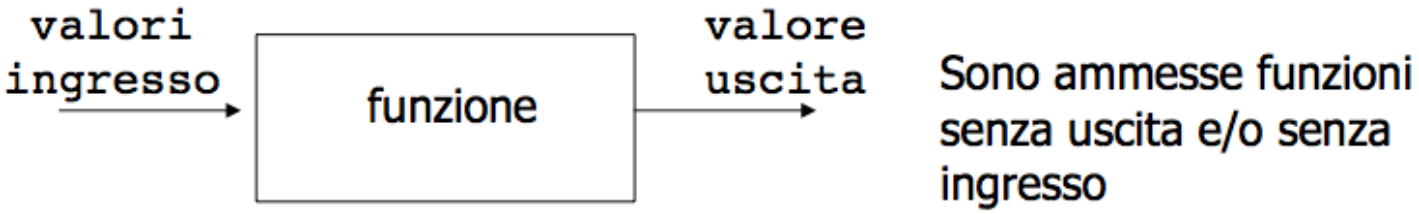
Programma con una funzione *square* per calcolare il quadrato di un numero.

```
#include<stdio.h>
int square(int); /*dichiarazione di funzione (o prototipo)*/

int main() {
   int x;

   for(x=1;x<=10;x++)
      printf("il numero %d quadrato di %d \n",square(x),x);

   return 0;

}
int square(int y) {/* definizione della funzione */
   return y * y;
}
```

Outline

Dichiarazione di Funzioni

Come le variabili devono essere dichiarate prima di poter essere usate, anche le funzioni necessitano di una *dichiarazione*.



La dichiarazione di una funzione dovrà specificare:

1. Il nome della funzione: identificatore che ne permetterà l'utilizzo nel programma;
2. Gli argomenti (**parametri formali**) della funzione: quantità e tipo dei valori che verranno forniti in ingresso;
3. Il tipo del valore in uscita.

Sintassi della dichiarazione

La sintassi della dichiarazione di funzione (*prototipo*) è la seguente:

```
identificatore-tipo identificatore(lista-tipo-parametri-formali);
```

`Identificatore-tipo` specifica il tipo del valore del risultato. `Identificatore` specifica il nome della funzione. `Lista-tipo-parametri-formali` specifica il tipo dei parametri di input: tipo1 nome1, tipo2, nome2, . . ., tipoN nomeN. Il nome dei parametri nel prototipo è facoltativo.

Esempi di dichiarazione

Prototipi:

```
int fattoriale(int);     oppure
int fattoriale(int n);

int mcd(int, int);       oppure
int mcd(int a, int b);

double lit2euro(int);    oppure
double lit2euro(int lire);
```

Il nome del parametro può essere incluso nel prototipo per documentare meglio il programma, comunque sarà ignorato dal compilatore.

Tipo void

Esistono funzioni che non producono alcun effetto (es. funzione di stampa)
Il linguaggio C mette a disposizione un tipo speciale *void*:

```
void f(int,int);
```

E' il prototipo di una funzione che non restituisce nessun output. Per esempio la sua definzione può essere la seguente:

```
void f(int a, int b) {
        printf("%d",a*b);
    }
```

Il tipo void viene utilizzato anche per specificare l'assenza di argomenti: le dichiarazioni int f(void); e int f(); sono equivalenti.

Definizione di Funzioni

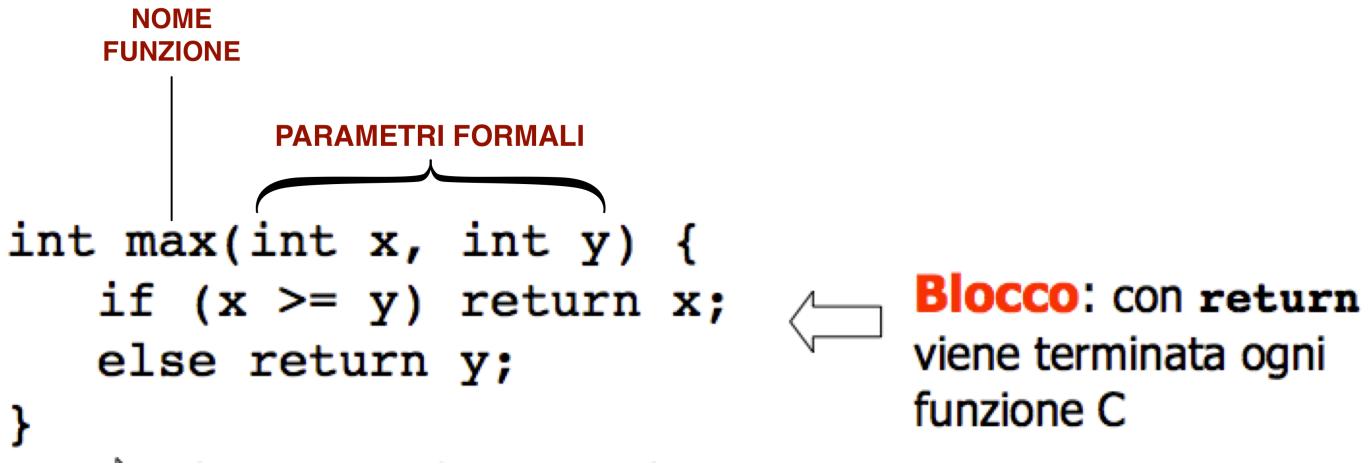
La definizione di una funzione specifica cosa fa una funzione e come lo fa: contiene la sequenza di istruzioni che dovrà essere eseguita per ottenere il valore risultato a partire dai dati in ingresso (una sorta di sottoprogramma).

La definizione di una funzione consta di due parti fondamentali:

1. **Intestazione**: simile alla dichiarazione (però necessaria la specifica dei nomi dei parametri formali);
2. **Blocco**: del tutto simile al blocco già introdotto per la funzione main() e può contenere istruzioni e dichiarazioni di variabili (nota: non di funzioni!).

Esempio di definizione

Vediamo un esempio semplice di definizione di funzione in C:



Attenzione: se si omette di indicare il tipo del parametro formale questo viene assunto `int`.

Sintassi della definizione

La sintassi di una dichiarazione di funzione è la seguente: `intestazione blocco`.
Blocco è il corpo della funzione (codice). L'intestazione ha la forma seguente:

```
identificatore-tipo identificatore ( lista-parametri-formali )
```

`identificatore − tipo` specifica il tipo del valore di ritorno, se manca viene assunto `int`. `identificatore` specifica il nome della funzione. `lista − parametri − formali` specifica una lista di dichiarazioni di parametri (tipo nome) separate da virgola (la lista può essere vuota).

Esempio di definizione di una funzione

```
/* trovare il maggiore di tre interi */
#include <stdio.h>

int maximum(int, int, int);

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

   printf("digita tre interi: ");
   scanf("%d%d%d",&a,&b,&c);
   printf("il massimo è: %d",maximum(a,b,c));
   return 0;
}
int  maximum(int x, int y, int z){
   int max = x;

   if (y > max) max = y;
   if (z > max) max = z;
   return max;
}
```

intestazione

blocco

Outline

Attivazione di funzioni

Le funzioni vengono chiamate (*attivate*) all'interno di funzioni.

```
  . . .
int valore, x, y;
  . . .
x = 1; y = 2;
valore = max(x,y);
  . . .
```

`x` e `y` sono i parametri attuali

Sintassi di attivazione di una funzione:

```
identificatore (lista-parametri-attuali)
```

`Identificatore` è il nome della funzione.
`lista − parametri − attuali` è una lista di espressioni separate da virgola. I parametri attuali devono corrispondere in numero e tipo ai parametri formali.

Conversione dei parametri

Una chiamata di funzione che non corrisponda al prototipo provocherà un errore di sintassi.

```
Es. int x;
    x = maximum(1,3);
```

Errore a tempo di compilazione

Se una conversione implicita negli argomenti è possibile questo viene fatto automaticamente, analogamente al caso dell'assegnamento visto in precedenza.

Esempio: `sqrt` ha come argomento un `double`, se però:

```
int x = 4;
printf("%f",sqrt(x)); stampa 2.00…0
```

Il compilatore promuove il valore intero 4 al valore double 4.0 quindi la funzione viene chiamata correttamente.

Semantica dell'attivazione

Semantica della *Attivazione Funzione* di una funzione *B* in una funzione *A*:

1. L'attivazione di una funzione è un'espressione;
2. L'attivazione di *B* da *A* determina:
   - viene sospesa l'esecuzione di *A* e si passa ad eseguire le istruzioni di *B* (a partire dalla prima).

Prima di poter essere attivata una funzione deve essere definita (o dichiarata).

Passaggio dei parametri (I)

Abbiamo visto che le funzioni utilizzano **parametri**

- permettono uno **scambio di dati** tra chiamante e chiamato
- nell'intestazione/prototipo: lista di parametri formali (con tipo associato) – sono delle variabili
- nell'attivazione: lista di parametri attuali — possono essere delle espressioni
- Al momento della chiamata ogni parametro formale viene inizializzato al valore del corrispondente parametro attuale.
- Il valore del parametro attuale viene **copiato** nella locazione di memoria del corrispondente parametro formale.
- Questo meccanismo di passaggio dei parametri viene comunemente detto passaggio per valore.

Passaggio dei parametri (III)

L'effetto della chiamata `succ()` può essere **simulato** dall'esecuzione della seguente porzione di codice:

```
succ(int x)  w = succ(20);     x = 20;
{x=x+1;                        x = x + 1;
return x}                      return x;
```

Passaggio dei parametri (III)

L'effetto della chiamata `succ()` può essere **simulato** dall'esecuzione della seguente porzione di codice:

```
succ(int x)  w = succ(20);     x = 20;
{x=x+1;                        x = x + 1;
return x}                      return x;
```

Alla fine **w** è **21**.

Passaggio dei parametri (III)

L'effetto della chiamata `succ()` può essere **simulato** dall'esecuzione della seguente porzione di codice:

```
succ(int x)  w = succ(20);     x = 20;
{x=x+1;                        x = x + 1;
return x}                      return x;
```

Alla fine **w** è **21**. Chiamate diverse corrispondono ad inizializzazioni diverse dei parametri formali

Passaggio dei parametri (III)

L'effetto della chiamata `succ()` può essere **simulato** dall'esecuzione della seguente porzione di codice:

```
succ(int x)  w = succ(20);     x = 20;
{x=x+1;                        x = x + 1;
return x}                      return x;
```

Alla fine **w** è **21**. Chiamate diverse corrispondono ad inizializzazioni diverse dei parametri formali

```
z = 10;
w = succ(z+3);     x = 13;
                   x = x + 1;
                   return x;
```

Passaggio dei parametri (III)

L'effetto della chiamata `succ()` può essere **simulato** dall'esecuzione della seguente porzione di codice:

```
succ(int x)
{x=x+1;
return x}
```

```
w = succ(20);
```

```
x = 20;
x = x + 1;
return x;
```

Alla fine **w** è **21**. Chiamate diverse corrispondono ad inizializzazioni diverse dei parametri formali

```
z = 10;
w = succ(z+3);
```

```
x = 13;
x = x + 1;
return x;
```

Alla fine **w** è **14**.

Il passaggio dei parametri di tipo array non comporta la copia dei valori dell'array

Dove si definisce una funzione

In C non si possono definire funzioni dentro funzioni. Quindi si definiscono fuori dalla funzione main() (sopra o sotto).

Esempio:

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

int main {
   int a, b,c;

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

definizione di funzione

chiamata di funzione

La funzione `max` è visibile nel `main`

Outline

Visibilità delle funzioni

Prima di essere attivata una funzione deve essere definita o dichiarata. Il compilatore deve controllare che i *parametri formali* corrispondano (per numero e tipo) ai *parametri attuali*. Prima dell'attivazione deve essere conosciuto il **prototipo**:

Esempio:

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

   scanf("%d%d",&a,&b);
   c = max(a,b);
   prinf("il max è %d",c);
   return 0;
}
int max(int x, int y) {
   if (x >= y) return x;
   else return y;
}
```

max non è visibile ed il compilatore C non può controllare i parametri

Visibilità delle funzioni (2)

Se la funzione non è definita almeno deve essere già dichiarata:

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

    scanf("%d%d",&a,&b);
    c = max(a,b);
    prinf("il max è %d",c);
    return 0;
}
int max(int x, int y) {
    if (x >= y) return x;
    else return y
}
```

dichiarazione di funzione

Ordine di dichiarazione

Ogni funzione deve essere stata dichiarata prima di essere usata. E' pratica comune specificare in quest'ordine:

1. Dichiarazione di tutte le funzioni con esclusione del `main`;
2. Definizione del `main`;
3. Definizione di tutte le altre funzioni.

In questo modo ogni funzione è stata dichiarata prima di essere usata:

Esempio:

```
int max(int,int);
int mcm(int,int);
int main(){ …}
int max(int a, int b){ … }
int mcm(int a, int b){…}
```

File di intestazione (Header)

Ogni libreria standard ha un corrispondente file header che contiene:

- definizioni di costanti;
- definizioni di tipo;
- dichiarazioni di tutte le funzioni della libreria.

Esempi:

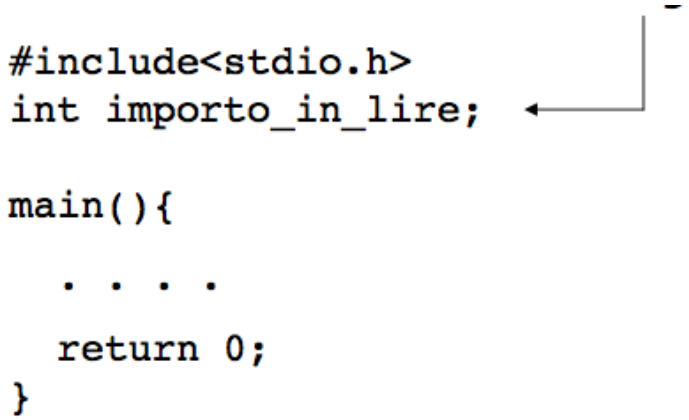
`<stdio.h>`	input/output
`<stdlib.h>`	memoria, numeri casuali, utilità generali
`<string.h>`	manipolazione stringhe
`<limits.h>`	limiti del sistema per valori interi
`<float.h>`	limiti del sistema per valori reali
`<math.h>`	funzioni matematiche
...	

Possono essere scritte dal programmatore "mialib.h".

Regole di visibilità (1)

Ogni dichiarazione del linguaggio C ha un suo campo di visibilità:

- Un identificatore **globale** viene dichiarato all'esterno di ogni funzione e sarà noto in tutte le funzioni definite successivamente;



- Un identificatore dichiarato all'interno del blocco è detto **locale** al blocco e sarà visibile in quel blocco e in tutti i blocchi in esso contenuti.

Regole di visibilità (2)

La visibilità di un identificatore locale è limitata al blocco

```
int a;
scanf("%d",&a);
if (a > 10) {
   int b = 10;
}
printf("%d",b);
```

Frammento di codice errato
(errore in fase di compilazione)

Il corpo di una funzione è un blocco: le sue ***variabili locali*** (che come vedremo sono allocate nello stack) sono riferibili solo nel corpo della funzione.

```
#include<stdio.h>
main(){
   int importo_in_lire;
   . . . .
   return 0;
}
```

variabile locale alla funzione `main`

In un blocco i nomi delle variabili devono essere tutti diversi.

Regole di visibilità (3)

I **parametri formali** di una funzione fanno parte dell'ambiente locale del blocco: visibili solo nel blocco della funzione.
Nota: non definire parametri con stesso identificatore usato per variabili locali.

```
int f(double x) {
    int x;
    . . .
  return 0;
}
```

Errore a tempo di compilazione

Mascheramento

Il mascheramento si ha quando in un blocco si dichiara un identificatore già dichiarato fuori dal blocco ed in esso visibile. La dichiarazione interna maschera quella esterna:

Esempio:

```
#include<stdio.h>
int importo_in_lire;

main(){
  double importo_in_lire;
     . . . .

  return 0;
}
```

```
int x1=10, x2=20;
char c='a';
int f(int);
main(){
 int x1=30;     /*nasconde la variabile globale x1 */
 x2 = x1+x2;    /*x1 e' quella locale, x2 e' globale */
 printf("x1=%d   x2=%d\n", x1, x2);   /* x1=30 x2=50*/
  {
    int x3=50;
    x1=f(x3); /*x1 e' quella locale al primo blocco */
    printf("x1=%d   x2=%d\n", x1, x2);  /* x1=150  x2=50
     */
  }
}
int f(int x1) {  /*nasconde la variabile globale x1 */
  int x2;        /*nasconde la variabile globale x2 */
  x2 = x1 + 100;  /*x1  p. formale, x2  var. locale*/
  return x2;
}
```

Tempo di vita di una variabile

Le **variabili locali** (le locazioni di memoria associate) vengono:

- Create al momento dell'attivazione di una funzione;
- Distrutte al momento dell'uscita dall'attivazione.

Da questo segue che:

- la funzione chiamante non può riferirsi ad una variabile locale alla chiamata;
- ad attivazioni successive di una stessa funzione corrispondono variabili (locazioni di memoria) diverse.

Nota: il tempo di vita di una variabile è un concetto rilevante a run-time.

Outline

Gestione della memoria virtuale a run-time

Codice macchina e dati entrambi in memoria principale, ma in zone separate:

- Memoria per il codice macchina fissato a tempo di compilazione;
- Memoria per dati locali alle funzioni (variabili e parametri) cresce e decresce dinamicamente durante l'esecuzione: gestito a pila (stack).

Una **pila** (o **stack**) è una struttura dati con accesso LIFO: Last In First Out = l'ultimo entrato è il primo ad uscire (es.: pila di piatti da lavare).

Gestione della memoria virtuale a run-time (II)

Il sistema gestisce in memoria la **pila dei record di attivazione** (RDA)

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

Ogni **RDA** contiene:

- le locazioni di memoria per i parametri formali (se presenti)
- le locazioni di memoria per le variabili locali (se presenti)
- altre informazioni che non analizziamo

Anche gli ambienti locali dei blocchi vengono allocati/deallocati sulla pila.

```
int f(int);
main()
{ int x, y, z;
   x=10;
   y=20;                /* ▸ PUNTO 1 : blocco principale */
   z = f(x);             /* ▸ PUNTO 2: prima chiamata di f */
     {
       int x=50; /* ▸ PUNTO 3: uscita da f  e iniz.  x */
       y=f(x);      /* ▸ PUNTO 4: seconda chiamata di f */
       z=y;           /* ▸ PUNTO 5: uscita da f */
     }
    ...                  /* ▸ PUNTO 6: uscita dal blocco */
}
int f(int a)
{ int z;
  z = a + 1;
  return z; }
```

Evoluzione della pila

PUNTO 1

x	10
y	20
z	?

▸ PUNTO 1

Evoluzione della pila

PUNTO 2

a	10
z	?

x	10
y	20
z	?

▸ PUNTO 2

Evoluzione della pila

PUNTO 3

x	50

x	10
y	20
z	11

▸ PUNTO 3

Evoluzione della pila

PUNTO 4

a	50
z	?

x	50

x	10
y	20
z	11

▸ PUNTO 4

Evoluzione della pila

PUNTO 5

x	50

x	10
y	51
z	11

▸ PUNTO 5

Evoluzione della pila

PUNTO 6

x	10
y	51
z	51

▸ PUNTO 6

Esercizi

Trovate 6 esercizi nella piattaforma di autovalutazione:

- Esercizio 1: Funzione Media;
- Esercizio 2: Funzione Multipli;
- Esercizio 3: Somma di Potenze;
- Esercizio 4: MCD e mcm;
- Esercizio 5: Calcolo Pi Greco;
- Esercizio 6: Numeri maggiori.

Esercizio 1: passaggio di array ad una funzione

Un array (identificato dall'indirizzo in memoria del suo primo elemento) può essere passato come parametro ad una funzione come qualsiasi altra variabile

Esercizio 1: passaggio di array ad una funzione

Un array (identificato dall'indirizzo in memoria del suo primo elemento) può essere passato come parametro ad una funzione come qualsiasi altra variabile

- Esempio: `funzione(int array[],...);`
- Oppure: `funzione(int* array,...);`