ISIS "Guido Tassinari" di Pozzuoli

Indirizzo INFORMATICO - Articolazione INFORMATICA

=B: CFA5H=75

http://www.riochierego/informatica.htm

A.S. 2012/2013

Docente: **F]c 7\ ]YfY[c**

LINGUAGGIO C
Funzioni e procedure

Sommario

- Introduzione
- Funzioni in C
  - Intestazione
  - Variabili interne ed esterne
  - Regole di visibilità
- Procedure in C
- Funzioni ricorsive (cenni)

Introduzione

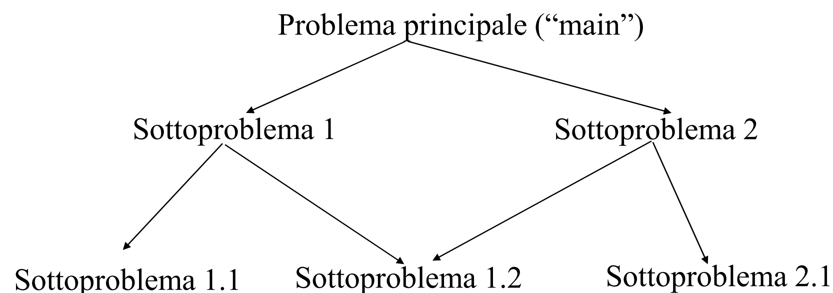
- Finora abbiamo visto le strutture di controllo di base che il C mette a disposizione per alterare il flusso di esecuzione delle istruzioni
- Da questo punto di vista, le funzioni e le procedure costituiscono un metodo particolarmente raffinato per il controllo del flusso di esecuzione
- Possiamo definire una **funzione** come un **raggruppamento di istruzioni volte a risolvere un determinato sottoproblema** entro il problema principale
  - Si pensi per esempio alle "funzioni di libreria" usate in precedenza
- La conoscenza e l'uso di tali strumenti fa la differenza fra un progettista di algoritmi (nonché programmatore) mediocre ed un vero "ingegnere del software"

Approcci di progYHUn]cbY

- In generale, in problemi complessi possono essere "individuati" sottoproblemi, la cui soluzione ad esempio è necessario calcolare molte volte all'interno dell'algoritmo, ma con "ingressi" differenti
- Questo conduce a due sostanziali approcci (strategie) per la risoluzione dei problemi:
  - Top-Down
  - Bottom-Up

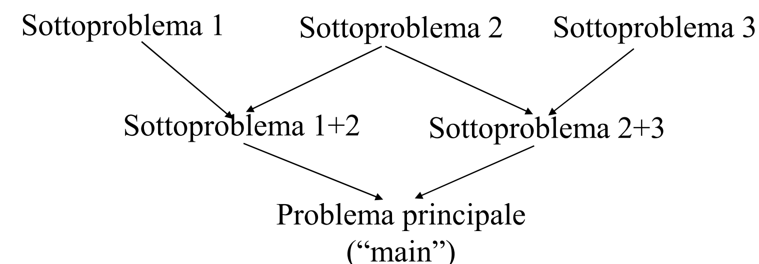
L’approccio Top-Down

- Si parte dal problema principale e via via si individuano i sottoproblemi che lo compongono



L’approccio Bottom-Up

- Si individuano problemi essenziali, il più semplici possibile, che, integrati gradualmente, permettono la soluzione del problema principale



L’approccio “sandwich”

- In generale è difficile seguire il Top-Down od il Bottom-Up in modo rigoroso, in quanto
  - certe funzionalità possono risultare utili in un secondo momento
  - non si ha un’idea chiara di come il problema principale vada risolto nella sua integrità ma si è riusciti ad individuare alcune funzionalità di base
- Si preferisce un approccio “ibrido”, chiamato anche “sandwich”, in cui le due strategie vengono condotte in parallelo
  - Ad es. su una parte del problema si segue la Top-Down, su altre la Bottom-Up
- La convergenza delle due strategie porta alla soluzione del problema
- **Si noti comunque come, in tutti i casi, la capacità di scomporre il problema in problemi più semplici sia essenziale ➔ MODULARITA’ DEL SOFTWARE**
  - si migliora la chiarezza del programma
  - ne si attenua la “rigidità”

Dfc[fUa a Un]cbY: un`YgYa d]c`gYa d`]WY sull'utilizzo dei sottoprogrammi

- Scrivere un programma C che, leggendo da tastiera due valori qualsiasi a e b , calcoli la loro somma s e la stampi a video
- La formula per il calcolo di s è la seguente:

  s = a + b

Algoritmo: soluzione di “alto livello”

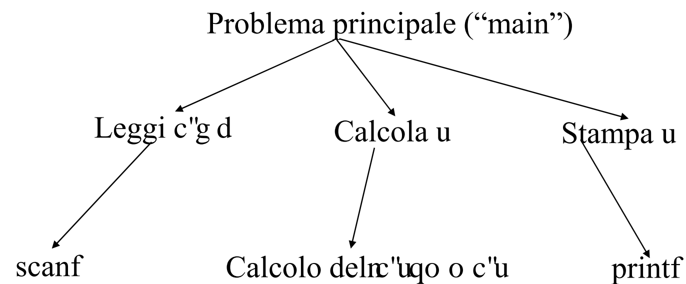
- Leggi due valori qualsiasi da tastiera a e b
- Calcola s
- Stampa a video s

Algoritmo: individuazione delle funzionalità

- Leggi due valori da tastiera a e b
  - Uso scanf
- Calcola s
  - uso la formula s = a + b
- Stampa a video s
  - Uso printf

Quindi...

- Siamo riusciti a scomporre il problema in sottoproblemi più elementari
- Resta da capire come scrivere “Calcola della somma s”



Funzioni in C

- Una funzione in C è caratterizzata dalla seguente sintassi:

```
Tipo_in_uscita NOME_FUNZIONE(Lista_parametri)
{
  [Parte dichiarativa;]
  Corpo della funzione;
  return Variabile di tipo Tipo_in_uscita;
}
```

Funzioni in C

- `Tipo_in_uscita` corrisponde al tipo di dato identificato come valore calcolato dalla funzione === rappresentazione della soluzione del sottoproblema

```
Tipo_in_uscita NOME_FUNZIONE(Lista_parametri)
{
  [Parte dichiarativa;]
  Corpo della funzione;
  return Variabile di tipo Tipo_in_uscita;
}
```

`N.B. Tipo_in_uscita = void per le procedure`

Funzioni in C

- `NOME_FUNZIONE` identifica il sottoproblema che essa risolve (es. `Calcola_Fattoriale`):

```
Tipo_in_uscita NOME_FUNZIONE(Lista_parametri)
{
  [Parte dichiarativa;]
  Corpo della funzione;
  return Variabile di tipo Tipo_in_uscita;
}
```


Funzioni in C

- `Lista_parametri` è una lista di variabili che vengono fornite alla funzione (parametri di ingresso) ed eventualmente altre che la funzione fornisce (parametri di uscita)

```
Tipo_in_uscita NOME_FUNZIONE(Lista_parametri)
{
  [Parte dichiarativa;]
  Corpo della funzione;
  return Variabile di tipo Tipo_in_uscita;
}
```


Parametri di ingresso e uscita

- `Lista_parametri` presenta questa forma
  - `Lista_parametri = Tipo1[*] NomeVariabile1, Tipo2[*] NomeVariabile2, …, TipoN[*] NomeVariabileN`
- Alcune di queste variabili vengono “prestate” alla funzione dall’esterno (es. dalla parte “principale” del programma) e vengono usate per calcolare l’uscita senza essere modificat

➔ parametri di ingresso(passati per valore)

- Altre possono fare “parte” della soluzione del sottoproblema e quindi essere modificate dopo l’esecuzione dell’ultima istruzione della funzione **`return`**

➔ parametri di uscita (per riferimento o per indirizzo)

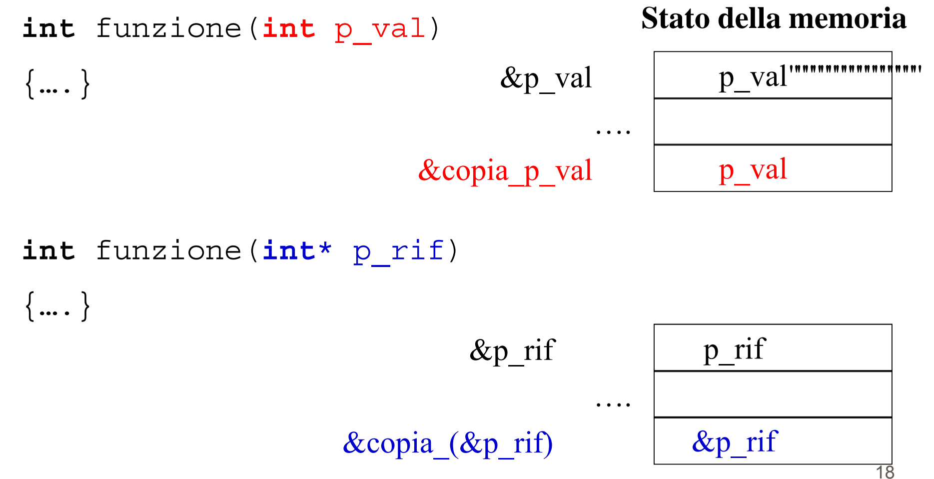
- Le parentesi quadre [*] indicano un elemento, l’asterisco, *eventualmente* aggiunto dopo la dichiarazione di tipo
- La `Lista_parametri` può anche essere vuota, in tal caso, tra parentesi si scrive semplicemente il tipo “non definito” **`void`**:
  - `Lista_parametri = void`

Passaggio XY]˙dUfUa Yhf]

- Un parametro di ingresso viene tipicamente passato “per valore”: il C fa una copia del contenuto della variabile e all’interno usa quella, senza alterarne quindi il contenuto quando la funzione termina
- Un parametro di uscita [Á§*¦^••[Ð•&ããÁ viene invec^ passato “perÁriferimento”: il contenuto della variabile è accessibile allaÁfunzione, e può essere modificato, in quanto vieneÁpassato *l’indirizzo* della variabile
  - Ciò si compie aggiungendo un asterisco dopo la dichiarazione di tipo:
  - Es. `Tipo1* NomeVariabile1`

```
N.B. FANNO ECCEZIONE GLI ARRAY CHE VENGONO PASSATI
     SEMPRE PER RIFERIMENTO
```


Differenza tra parametro per valore e per variabile



Funzioni in C

- Dopo la graffa aperta, vi è eventualmente una `Parte dichiarativa` in cui possono essere dichiarate appunto delle **variabili locali** della funzione.
  - Variabili locali: esse non sono “visibili” all’esterno in nessun modo.
  - La `Parte dichiarativa` comprende la dichiarazione della variabile che verrà restituita in uscita attraverso l’istruzione **`return`**

```
Tipo_in_uscita NOME_FUNZIONE(Lista_parametri)
{
  [Parte dichiarativa;]
  Corpo della funzione;
  return Variabile di tipo Tipo_in_uscita;
}
```


Funzioni in C

- Il `Corpo della funzione` è il codice che serve per risolvere il sottoproblema, che elabora le variabili nella `Lista_parametri` servendosi delle variabili dichiarate nella `Parte dichiarativa`

```
Tipo_in_uscita NOME_FUNZIONE(Lista_parametri)
{
  Parte dichiarativa;
  Corpo della funzione;
  return Variabile di tipo Tipo_in_uscita;
}
```

Procedure in C

- Sono esattamente equivalenti alle funzioni, con l'unica differenza che `Tipo_in_uscita` è sempre **`void`**
- La soluzione del sottoproblema è rappresentata dall'insieme dei parametri di uscita
- L'ultima istruzione **`return`** può anche essere omessa

```
void NOME_PROCEDURA(Lista_parametri)
{
  Parte dichiarativa;
  Corpo della procedura;
  return;
}
```


Torniamo al problema

- Procediamo in modo top-down, e scriviamo una prima versione del programma

```
/*Programma per il calcolo della somma*/
#include <stdio.h>
int main(void)
{
int a,b,s;
/*Leggi a, b da tastiera*/
..............
/*Calcola s*/
..............
/*Stampa s a video*/
..............
return 0;
}
```


Leggi U, V da tastiera: procedura `Leggi`

- Scriviamo una procedura che legga da tastiera due interi e li memorizzi in N e K, passati in ingresso come parametri

```
void Leggi( int* a, int* b)
{
printf("Inserisci  i  valori di a  e b :\n");
scanf("%d",a);   /* a contiene un indirizzo */
scanf("%d",b);   /* b contiene un indirizzo */
return;
}
```


Stampa s a video: procedura `Stampa`

- Soluzione banalissima:

```
void Stampa( int s)
{
  printf("Il valore richiesto  è:%d\n", s);
  return;
}
```

Calcola s: W¢b˙funzione SommaF

- Scriviamo una funzione che restituisca il valore richiesto:

```
int SommaF ( int a, int b)
{
int s;

/*Calcola la somma di a e di b */
s = a + b;
return s;
}
```


Calcola s: W¢b˙procedura SommaP

- Sriviamo una procedura che restituisca il valore richiesto:

```
void SommaP ( int a, int b, int* s)
{
/*Calcola la somma di a e di b */
*s = a + b;
return;
}
```


Completamento del programma 1

- Sostituiamo ai commenti le **chiamate** alle funzioni con i relativi parametri

```
/*Programma che usa SommaF */
#include<stdio.h>
int main(void)
{
int a, b, s;

Leggi (&a, &b);
s = SommaF (a,b);
Stampa (s);

return 0;
}
```


Completamento del programma 2

- Completiamo il programma sostituendo ai commenti le opportune chiamate alle funzioni

```
/*Programma che usa SommaP */
#include <stdio.h>
int main(void)
{
int a, b, s;

Leggi (&a, &b);
SommaP (a,b,&s);
Stampa (s);
return 0;
}
```

Completamento dei programmi 1 e 2

- Se però scrivessimo su un editor i programmi delle slide precedenti, e tentassimo di eseguirlo, avremmo subito un errore in fase di *compilazione*:
  - il compilatore ci chiederebbe “dove sono le funzioni Leggi, SommaF, SommaP,Stampa perché io possa tradurle?”
- Le funzioni che abbiamo scritto possono essere inserite in tre modi:
  - Nello stesso file del programma **prima** della funzione `main (Soluzione 1)`
  - Nello stesso file del programma **dopo** la funzione `main`, ma facendo precedere a questa le intestazioni di tutte le funzioni, chiamate **prototipi di funzione (Soluzione 2)**
  - In uno più file a parte (Soluzione 3)

Soluzione 1: prima del `main`

```
/*Programma perla somma di due interi*/
#include <stdio.h>

void Leggi( int* a, int* b)
{
    printf("Inserisci i valori a e b:\n");
    scanf("%d",a);
    scanf("%d",b);
    return;
}

void Stampa( int s)
{
    printf("Il valore richiesto è: %d\n", s);
    return;
}

int SommaF( int a, int b)
{
    int s;  /*parte  dichiarativa */

    /*corpo funzione*/
     s = a + b;
    return s;
}

void SommaP( int a, int b, int* s)
{
    /*corpo procedura*/
    *s = a + b:;
    return s;
}


int main(void)
{
    int a,b,s;

    Leggi(&a,&b);
    s = SommaF(a,b);          /* FUNZIONE */
    SommaP (a, b, &s);        /* PROCEDURA */
    Stampa(s);
    return 0;
}
```


Soluzione 2: dopo il `main`

```
/*Programma per il calcolo combinatorio*/
#include <stdio.h>

void Leggi( int* a, int* b);
void Stampa(int s);
int SommaF (int a, int b);
void SommaP(int a, int b, int* s);

int main(void)
{
    int a, b, s;
    Leggi(&N,&K);
    s = SommaF(a,b);   /* FUNZIONE */
    SommaP(a,b, &s);   /* PROCEDURA */
    Stampa(s);
    return 0;
}

void Leggi (int* a,int* b)
{
    printf("Inserisci i valori a e b:\n");
    scanf("%d",a);
    scanf("%d",b);
    return;
}

void Stampa ( int s)
{
    printf("Il valore richiesto è: %d\n",s);
    return;
}

int SommaF (int a, int b)
{
    int s;
    s = a + b;

    return s;

}

void SommaP(int a, int b, int* s)
{
    *s = a + b;
    return;
}
```


Soluzione 3: in uno o più file a parte

```
FILE PRINCIPALE "somma.c"
/*Programma perla somma di due interi*/
#include "funzioni.c"

int main(void)
{
    int a, b, s;
    Leggi(&a,&b);
    s = SommaF(a,b);     /* FUNZIONE */
    SommaP (a, b, &s);   /* PROCEDURA */
    Stampa(s);
    return 0;
}

FILE "funzioni.c"
#include <stdio.h>

void Leggi(int* a, int* b);
void Stampa(int s);
int SommaF (int a, int b);
void SommaP(int a, int b, int* s);

void Leggi(int* a, int*b)
{
   printf("Inserisci i valori a e b:\n");
   scanf("%d",a);
    scanf("%d",b);
   return;
}

void Stampa( int s)
{
    printf("Il valore richiesto è: %d\n", s);
    return;
}

int SommaF (int a, int b)
{
    int s;
    s= a + b;
    return s;
}

void SommaP (int a, int b, int* s)
{
    *s = a + b;
    return s;
}
```

Il concetto di blocco

- Definizione:
  un **blocco** consiste di due parti sintattiche racchiuse tra parentesi graffe:
  - Una parte dichiarativa (facoltativa)
  - Una sequenza di istruzioni
- Diversi blocchi possono comparire internamente al `main` o alle *funzioni* che compongono un programma

Esempi di blocchi

```
#include <stdio.h>
int g1,g2;
int f1(int par1, int par2);
...
main()
{
int a,b;
...
        /* blocco1 */
  {
    char a,c;
     ...
  }
...
}
```

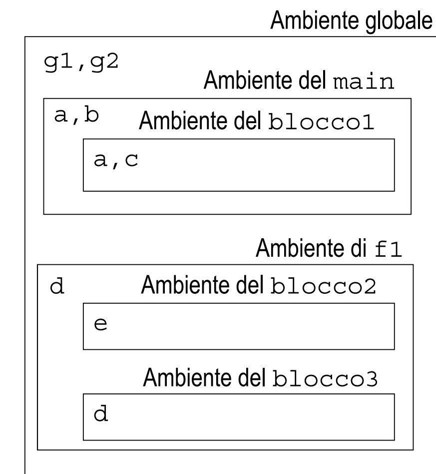
continua⇨

```
int f1(int par1, int par2)
{
int d;
...
  {           /* blocco2 */
    int e;
     ...
  }
  {           /* blocco3 */
    int d;
     ...
  }
...
}
```


Visibilità delle variabili

- Vediamo alcune definizioni:
  - Si dice **ambiente globale di un programma** l'insieme di tutti gli elementi dichiarati nella sua parte dichiarativa globale
  - Si dice **ambiente locale di una funzione** l'insieme di tutti gli elementi dichiarati nella sua parte dichiarativa e nella sua testata
  - Si dice **ambiente di un blocco** l'insieme di tutti gli elementi dichiarati nella sua parte dichiarativa
- Il concetto di ambiente permette di dichiarare più volte lo stesso identificatore anche con significati diversi, purché in ambienti diversi

Visibilità delle variabili – modello a contorni



Elementi nell'**ambiente globale** possono essere "visti" da tutte le funzioni e i blocchi

Elementi nell'**ambiente locale** di una funzione possono essere "visti" da tutti i blocchi contenuti nella funzione

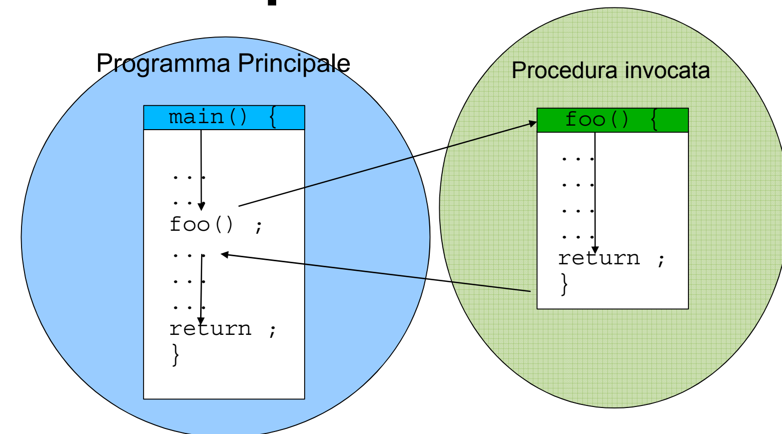
Gli elementi nell'**ambiente di un blocco** possono essere "visti" da tutte le istruzioni nel blocco e dai blocchi in esso contenuti

In caso di più definizioni dello stesso identificatore:
è valida la definizione dell'ambiente più vicino al punto di utilizzo

Tempo di vita delle variabili

- Creazione di una variabile:
  allocazione in memoria dello spazio per la sua rappresentazione
- Due categorie di variabili:
  - **Variabili fisse o statiche:**
    allocate una sola volta e vengono distrutte solo quando termina l'esecuzione del programma (es. variabili globali del programma)
  - **Variabili automatiche:**
    create ogni volta che si entra nel loro ambito di visibilità e che vengono distrutte all'uscita da tale ambiente (es. variabili dichiarate a livello di funzione, procedura e blocco)

La macchina virtuale C e l'esecuzione di funzioni o procedure



Concettualmente si può modellare la relazione tra programma principale e procedura invocata con la generazione di un macchina virtuale C per ogni procedura, con un proprio "ambiente locale" ed una parte "condivisa" con la macchina virtuale chiamante

Passaggio di vettori come parametri

- Il passaggio di vettori è l'unico caso in cui si è obbligati ad usare il passaggio per "riferimento"
- Il C passa alla funzione invocata solo l'indirizzo del primo elemento del vettore, quindi la manipolazione dei valori all'interno ne determina l'alterazione

Esempio 1: il vettore nella lista parametri

- Scrivere una procedura C che, ricevendo in ingresso un vettore di interi `v` e due valori `i` e `j`, permuti il valore in posizione `i` con quello in posizione `j`

```
/*Procedura per lo scambio di due valori in un vettore*/
void scambia(int* v, int i, int j)
{
  int temp;
  temp=v[i];
  v[i]=v[j];
  v[j]=temp;
  return;
}
```

Il vettore `v` viene passato anteponendo al suo identificatore simbolico il carattere '*' oppure scrivendo int v [ ]

Fondamenti di Informatica 1 - A.A. 2010/11 - Prof. Gian Luca Marcialis

Esempio 2: il vettore come tipo restituito

- Scrivere una funzione C che, ricevendo un vettore `v` di `N` elementi (`N` definito tramite `#define`), restituisca un secondo vettore `w` che presenti la seguente corrispondenza: `w[N-1] == v[0], w[N-2]== v[1], …, w[N-i] == v[i-1], …, w[0] == v[N-1]`

```
int* inverti(int*v)
{
  int i, w[N];
  for (i=0; i<N; i++)
     w[i]=v[N-(i+1)];
  return w; /*w si può scrivere anche &w[0]*/
}
```


Esercizio

- Scrivere una funzione `media_mobile` C che, ricevendo in ingresso un vettore `v` di `N` interi, restituisca un vettore `w` (di `N` interi) tale che:
  - `w[i] = (v[i] + v[i+1])/2;`
    - per `i=0,…,N-2`
  - `w[N-1] = (v[N-1] + v[0])/2`

Soluzione

```
int* media_mobile(int*v)
{
  int w[N];

  for(int i=0; i<=N-2; i++)
     w[i]=(v[i]+v[i+1])/2;

  w[N-1]=(v[N-1]+v[0])/2;

  return w;
}
```


Esercizio

- Scrivere una funzione C `somma` che, ricevendo in ingresso due vettori di interi `v` e `w` di dimensione `N` predefinita (es. attraverso `#define`) restituisca un vettore `z` tale che:
  - `z[0] = v[0] + w[N-1]`
  - `z[1] = v[1] + w[N-2]`
  - …
  - `z[N-1] = v[N-1] + w[0]`

Funzioni (e procedure) ricorsive

- Si dice che una funzione è ricorsiva quando richiama sé stessa al suo interno
- Non c'è alcun conflitto tra le variabili ed i parametri perché ogni chiamata crea una macchina virtuale (un ambiente) indipendente
- Naturalmente bisogna fare attenzione alle funzioni ricorsive in cui alcuni parametri sono passati per variabile

Esempio: la funzione `fattoriale`

- La definizione di fattoriale si può scrivere:
  - fattoriale(0) = 1;
  - fattoriale(n) = n*fattoriale(n-1)
- E' un esempio classico di definizione ricorsiva, la cui implementazione C è:

```
unsigned int fattoriale(unsigned int j)
{
  if(j==0)
      return 1;
  else
      return j*fattoriale(j-1);
}
```


Scrivere funzioni ricorsive

- Si deve individuare il **passo radice** della funzione ricorsiva, che pone termine alla ricorsione
- Si invoca la funzione ricorsivamente se il passo base non viene verificato

```
unsigned int fattoriale(unsigned int j)
{
  if(j==0)
      return 1;
  else
      return j*fattoriale(j-1);
}
```


Esercizio

- Scrivere una funzione ricorsiva `eleva` che, ricevendo in ingresso due interi senza segno `x` e `y`, restituisce x^y.
- Definizione ricorsiva di elevamento a potenza:
  - $x^y = 1$, se y==0;
  - $x^y = x * x^{y-1}$, se y>0

```
unsigned int eleva(unsigned int x, unsigned int y)
{
  if(y==0)
      return 1;
  else
      return x*eleva(x,y-1);
}
```

Nota

- Per ogni funzione ricorsiva, esiste il suo corrispondente NON ricorsivo
- Scrivere funzioni ricorsive rende più leggibile il programma, ma in generale ne rallenta l'esecuzione
- Ci sono inoltre situazioni in cui la ricorsione è sconsigliata, soprattutto quando sono previsti molti passi di ricorsione all'interno della procedura
  - La generazione di una macchina virtuale C corrisponde infatti, nella pratica, all'allocazione di nuove aree di memoria per ogni chiamata ricorsiva
  - Tali aree sono tanto più "grandi" quanto maggiori sono le risorse (variabili, parametri etc) richieste dalla funzione stessa
  - Se le chiamate eccedono un certo numero, potrebbe non esserci più memoria per la generazione di una nuova macchina virtuale
- Si provi ad esempio la funzione `fattoriale` ricorsiva con `j` relativamente "grande" e si confronti il comportamento con il corrispondente non ricorsivo

Per saperne di più

- Ceri, Mandriola, Sbattella, *Informatica – arte e mestiere*, Capo. 7-8, McGraw-Hill
- Kernighan, Ritchie, *Il linguaggio C*, Cap. 4, Pearson-Prentice Hall