UNIVERSITÀ
degli STUDI
di CATANIA

Static, inline, Ricerca e ricorsione

Corso di programmazione I (A-E / O-Z) AA 2022/23

Corso di Laurea Triennale in Informatica

Fabrizio Messina
fabrizio.messina@unict.it

Dipartimento di Matematica e Informatica

Keyword static

Varibili static.

```
1 void foo(){
2     static int foo_calls = 0;
3     foo_calls++;
4     printf("foo_calls: %d", foo_calls);
5 }
```

La variabile static viene allocata con la prima chiamata a funzione.

La variabile static viene deallocata solo alla fine del programma.

La variabile static `foo_calls` viene inizializzata solo alla prima invocazione della funzione.

Keyword static

Funzioni static

```
1 // main.c
2 static void foo(){  return; }
3
4 int main(){  foo(); bar(); }
5
6 // bar.c
7 void bar(){ foo(); /* NO!! */ }
```

La funzione static potrà essere invocata solo da codice appartenente allo stesso file sorgente (translation unit).

In altre parole, la visibilità della funzione non sarà più globale ma "locale" alle sole funzioni presenti nel suo modulo sorgente.

Keyword static

Esempi Svolti

`18_static_var.c`

`18_static_functions.c`, `18_bar.c`

Keyword extern

Dichiarare una funzione extern: funzione (non static!) che è definita in un altro file sorgente.

```
1 //main.c
2 extern void foo();
3
4 int main(){
5     foo(); // call to global function foo()
6 }
```

- la dichiarazione dei prototipi avviene nei file header
- anche se non si inserisce `extern`, il compilatore "assume" che la funzione sia "extern".

Keyword extern

Variabili extern

Una variabile definita fuori dalle funzioni avrà visibilità globale.

Tuttavia va dichiarata con la parola chiave **extern** in ogni modulo diverso da quello in cui la variabile stessa è definita.

```
1 //zoo.c
2 int var = 10;
3
4 void zoo(){ int a = var*2; }
5
6 //main.c
7 extern int var;
8 int main(){ int x = var*3; }
```

Keyword extern

```
1 //zoo.c
2 int var = 10;  //dichiarazione e definizione
3
4 //main.c
5 extern int var; // solo dichiarazione
```

In assenza della dichiarazione `extern int var`, il compilatore produrrebbe un errore.

Nella pratica, la dichiarazione di tali variabili extern andrebbe nei file header.

La dichiarazione di una variabile extern non implica allocazione di memoria.

Keyword extern

Esempi Svolti

`18_extern_var.c`, `18_zoo.c`

Keyword inline per le funzioni

```
1   int sum(int a, int b){
2     return a+b;
3   }
```

Il vantaggio evidente dell'uso delle funzioni e' il **riuso del codice e la modularità.**

Tuttavia, chiamate a funzioni costose in termini di tempo di cpu.

Keyword inline per le funzioni

L'introduzione della keyword `inline` in C++ (ed in C):

- permette al programmatore di "suggerire" al compilatore di sostituire le chiamate ad una certa funzione con il corpo della funzione stessa;
- alternativa all'uso di macro (preprocessore);
- C e C++ differiscono nell'interpretazione della keyword "inline";

NB: I compilatori scelgono di effettuare "inlining" del corpo della funzioni indipendentemente dai suggerimenti del programmatore, per ottenere codice più veloce..

Keyword inline per le funzioni

1) MACRO (preprocessore)

```
1 #define inc(i) (i+1);
```

VS

2) Uso di inline (C) (perchè anche static? Vedi prossima slide..)

```
1 static inline int inc(int i){ return i+1 };
```

```
1 int main(){
2 //..
3 printf("%d", inc(3));
```

Keyword inline per le funzioni

Semantica (C) della keyword inline:

- si suggerisce al compilatore di sostiture la chiamata alla funzione con il corpo della stessa;
- dato che la funzione è inline, la sua implementazione **non viene passata al linker**. Quindi:
  - nel caso il compilatore non decida di sostituire la chiamata a funzione con l'implementazione inline suggerita dal programmatore, il compilatore produrrà una apposita chiamata
  - il linker produrrà **errore: undefined reference to function..**
- Di conseguenza, **va usata la keyword** `static` per "forzare" il compilatore a usare, all'interno del modulo, la definizione inline della funzione.

Keyword inline per le funzioni

Nella pratica:

```
1 //utils.h
2 static inline int inc(int i){ return i+1 };
3
4 //foo.c
5 #include <utils.h>
6 void foo(){ int h = inc(3); }
7
8 //bar.c
9 #include <utils.h>
10 void bar(){ int z = inc(3)*4; }
```

Keyword inline per le funzioni

Esempi Svolti

`imain.c, ibar.c, ifoo.c, iutils.h`

Ricerca sequenziale di un elemento in un array.

Data una struttura dati lineare (esempio: array), ricercare un elemento X tra le componenti dello array.

Nella sua forma più semplice: (ricerca tutte le occorrenze del numero nella stuttura lineare).

```
1    int A[DIM];
2    //...
3    while (i++<DIM)
4    {
5       if(A[i] == numero)
6          printf("\n Found at index %d", i);
7    }
```

Esempi Svolti

`18_01.c` – Ricerca lineare di **tutti gli elementi**

`18_02.c` – Ricerca lineare, si ferma alla **prima occorrenza**

`18_03.c` – Ricerca lineare con **sentinella**

`18_04.c` – Ricerca lineare in **array ordinato**

`18_05.c` – Ricerca lineare del **massimo valore**

Ricerca dicotomica

Dicotomia == divisione in due parti.

Sia A un **array ordinato** (in modo crescente o decrescente).

Si confronta la chiave di ricerca con l'emento centrale (M) dello array :

- se sono uguali, l'elemento è stato trovato;
- se la chiave di ricerca è minore di M, la ricerca prosegue iterativamente nella **prima metà di A**;
- se la chiave di ricerca è maggiore di M, la ricerca prosegue nella **seconda metà di A**;

Quindi: la ricerca dicotomica opera, ad ogni iterazione, su un array che è **la metà di quello precedente**.

- NB: condizione necessaria è che l'array sia ordinato.

Esempi Svolti

`18_06.c` – Ricerca dicotomica.

Ricorsione

Una funzione si dice ricorsiva quando è definita in termini di se stessa, ovvero quando nel corpo della funzione sono presenti **chiamate alla funzione stessa** (funzioni ricorsive):

- uno o più **casi base** (condizioni di terminazione)
- un passo ricorsivo, che si basa su una o più **chiamate alla funzione stessa** ma su un **input "ridotto"**

Esempio di funzione matematica definita mediante **induzione** anche detta **relazione di ricorrenza**.

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

dunque:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n-1)! & \text{se } n > 0 \end{cases}$$

ESEMPIO: Funzione ricorsiva per il calcolo del fattoriale.

```
1   long fattoriale(int n){
2     if(n == 0)
3       return 1;
4     return
5       n * fattoriale(n-1); //chiamata ricorsiva
6   }
```

Caso base o condizione di **terminazione**: n=0;

Chiamata ricorsiva con input ridotto (n-1);

Dato un certo problema, una soluzione che faccia uso di una funzione **ricorsiva**:

(+) è intuitivamente **più semplice** da concepire;

(+) **minor numero di linee di codice**;

(-) consuma **molta memoria** rispetto ad una soluzione iterativa;

(-) consuma **molto tempo** rispetto ad una soluzione iterativa;

Il **consumo di memoria** è dovuto alla allocazione dei record di attivazione delo stack dovuti alla **sequenza di chiamate ricorsive,** una dopo l'altra.

Il **consumo di tempo** è dovuto all'allocazione dei record sullo stack, la copia dell'indirizzo di ritorno e della variabili locali.

Qualsiasi funzione ricorsiva si può sempre esprimere in forma non ricorsiva.

- Ricorsione **di coda**: la chiamata ricorsiva è l'ultima azione della funzione ricorsiva.
- Ricorsione **non di coda**: la ricorsione può essere eliminata con l'ausilio di uno **stack esterno**.

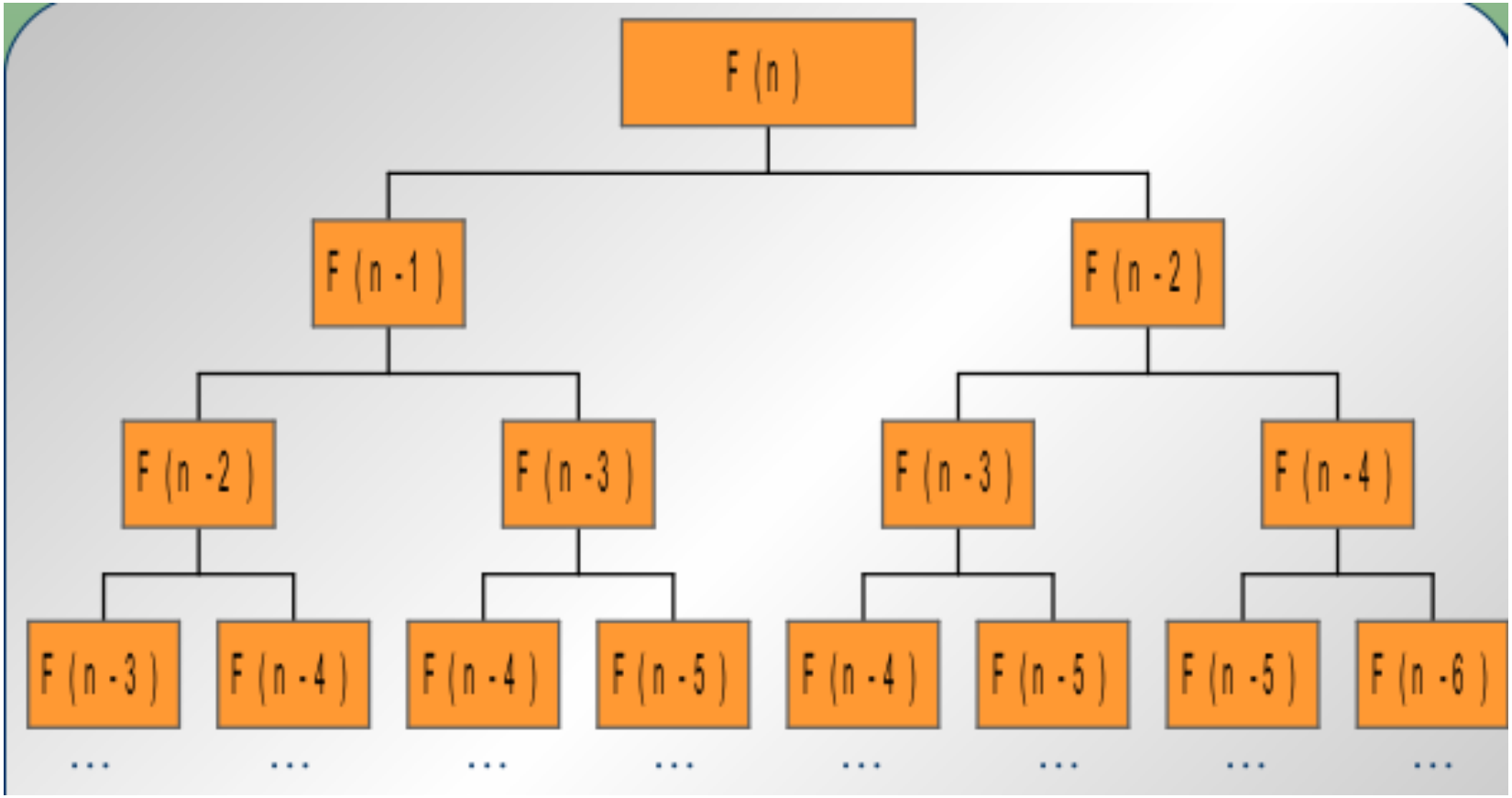
Anche le funzioni con una doppia chiamata ricorsiva **non si possono riscrivere** con una banale iterazione.

Numeri di Fibonacci

$F(0) = F(1) = 1;$
$F(n) = F(n\text{-}1) + F(n\text{-}2);$

```
1  int fibo(int n){
2    if(n<=1)
3      return 1;
4    else
5      return fibo(n-1) + fibo(n-2);
6  }
```

Inefficienza. (chiamate duplicate!)



Esempi Svolti

`18_07.c` – Fattoriale.

`18_09.c` – Ricerca dicotomica ricorsiva.

`18_10.c` – Numeri di Fibonacci.

Homework H18.1

Codificare una funzione per il calcolo del fattoriale di un numero intero con approccio **iterativo**.

Homework H18.2

Codificare una funzione per il calcolo dei numeri di fibonacci con approccio **iterativo**.

Homework H18.3

Codificare una funzione che calcoli la somma degli elementi di un array con approccio **ricorsivo**.

Homework H18.4

Codificare una funzione che prenda in input un numero che rappresenti una **base** ed un altro numero positivo che rappresenti **l'esponente**, e calcoli l'elevamento a potenza con approccio **ricorsivo**.

Riferimenti e approfondimenti

[1] → Capitolo 5 (5.14, 5.15, 5.16).
[1] → Capitolo 6 (6.10);

In particolare, esercizi sulla ricorsione proposti nei capitoli 5 e 6 del testo [1].

---

[1] Paul J. Deitel and Harvey M. Deitel.
C Fondamenti e tecniche di programmazione.
Pearson, 2022.