Ver. 3

© 2021 - Claudio Fornaro - Corso di programmazione in C (C89)

Struttura modulare

- Per semplificare la struttura di un programma complesso è possibile suddividerlo in moduli, detti anche *procedure*, *subroutine*, *sottoprogrammi* o, come in C, ***funzioni***
- Un modulo è un *blocco di codice* che assolve a un compito preciso (ad es. calcola la radice quadrata) e a cui è stato dato un *nome*
- Un programma consta di un modulo principale (in C è il `main`) ed eventuali altri moduli
- Il C è un *linguaggio procedurale* in quanto il suo paradigma di programmazione (organizzazione del codice) è basata sull'interazione tra moduli

Struttura modulare

- Quando un modulo richiama un altro modulo si ha che l’esecuzione del *modulo chiamante* viene sospesa finché il *modulo chiamato* non ha terminato la sua esecuzione
- Quando l’esecuzione del *modulo chiamato* è terminata, il *modulo chiamante* riprende la sua esecuzione **dal punto successivo** a dove è stata fatta la chiamata
- Ogni modulo può richiamare (far eseguire) qualsiasi altro modulo (in C può richiamare anche se stesso)

Struttura modulare

- Le due chiamate del modulo `StampaCiao` fanno eseguire ogni volta le istruzioni che lo costituiscono (sospendendo il chiamante)

Modulo chiamante (*es.* `main`)

```
...
scanf...
StampaCiao()
...
for ...
   switch ...
printf...
...
StampaCiao()
if (x==2) then
...
```

Prima chiamata

Modulo chiamato (**`StampaCiao`**)

```
printf("***  *   *   ***\n");
printf("*    *  *  *  *  *\n");
printf("*    *  ***  *  *\n");
printf("*    *  *  *  *  *\n");
printf("***  *  *  *  ***\n");
```

Seconda chiamata

Struttura modulare

- Vantaggi della programmazione modulare:
  - il programma complessivo ha un *maggior livello di astrazione* perché i moduli "nascondono" al loro interno i dettagli implementatativi delle funzionalità realizzate: il modulo è visto come un "qualcosa" da usare in un certo modo (valori passati e restituiti)
  - il codice per ottenere una certa funzionalità viene scritto una volta sola e viene richiamato ogni volta che è necessario
  - il codice complessivo è più corto
  - essendo più piccoli, i moduli sono più semplici da implementare e da verificare
  - il codice di un modulo correttamente funzionante può essere riutilizzato in altri programmi

Definizione di funzioni

- La sintassi della *definizione* di una funzione è la seguente:

  tipo_restituito ***nomeFunzione*** (*parametri*)
  {
  definizione_variabili_locali
  istruzioni
  eventuale `return`
  }

  corpo della funzione

- Il ***nomeFunzione*** è un normale identificatore
- Non si possono definire funzioni all'interno di altre funzioni: sono tutte allo stesso livello
- Per migliorare la leggibilità non si mettano spazi tra ***nomeFunzione*** e la parentesi

Corpo delle funzioni

- Il corpo della funzione può contenere:
  - la definizione di variabili locali (opzionale)
  - istruzioni di elaborazione
  - l'istruzione `return` (quando appropriato)
- La definizione delle variabili locali deve avvenire prima delle istruzioni di elaborazione
- Ai fini dello sviluppo del programma, il corpo di una funzione può essere temporaneamente vuoto (solo le parentesi con eventuale `return`), tale funzione viene chiamata *stub*)

Variabili locali

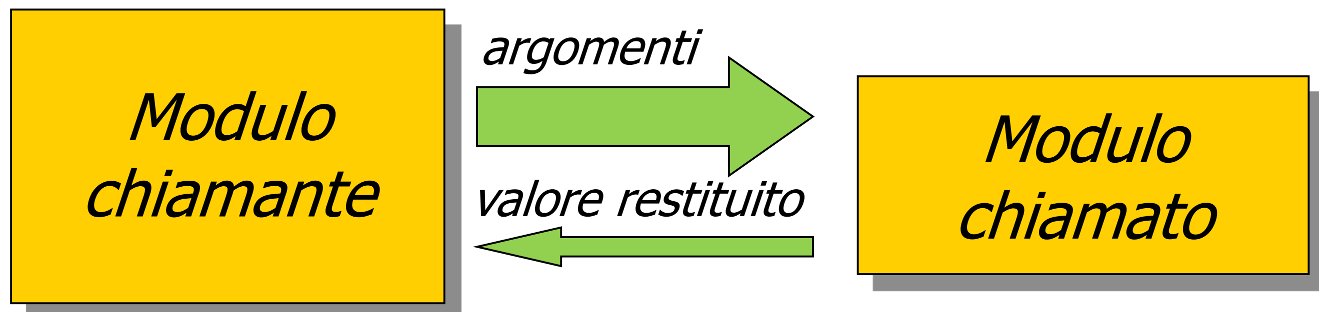
- Ogni funzione può essere considerata un piccolo programma *isolato* dalle altre funzioni (anche dal `main`)
- All'interno di una funzione possono essere definite ***variabili locali;*** essendo *locali* le altre funzioni non le "vedono" (si dice che hanno *scope locale* o *di blocco*, oppure che sono *private*)
- Variabili con lo stesso nome in funzioni diverse (compreso il `main`) sono tra loro del tutto indipendenti (per tipo e valore), in altre parole in memoria sono allocate in punti diversi

Variabili locali

- Il termine *storage duration* specifica quando nel tempo un oggetto è presente in memoria
- Le variabili locali hanno *storage duration* ***automatica:*** sono create ogni volta che la funzione viene chiamata e sono eliminate ogni volta questa termina (perdendone il valore)
- L'eventuale inizializzazione viene effettuata a ogni chiamata
- Senza inizializzazione il contenuto è indefinito
- Sono allocate sullo *stack* del processo (vedere dettagli più avanti)

Argomenti e valore restituito

- Essendo i moduli isolati, se il modulo chiamante deve passare dei valori da elaborare alla funzione (modulo chiamato) deve utilizzare particolari variabili dette *argomenti*
- La funzione può comunicare al modulo chiamante il risultato dell’elaborazione producendo (“restituendo”) un *unico* valore detto *valore restituito* o *valore di ritorno*



Chiamata di funzione

- Il chiamante chiama ("invoca") una funzione mediante il suo nome e indicando tra parentesi gli *argomenti* (separati da virgole):
```
eleva(y,2);
```
- Anche nel caso la funzione non richieda parametri, le parentesi devono sempre esserci:
```
StampaCiao();
```
- Il *valore restituito* può essere assegnato a una variabile, utilizzato in un'espressione o anche scartato (*expression statement*):
```
x = eleva(y,2);
y = 3*eleva(2,k)-4*k;
eleva(3,5); /* inutile */
```

Argomenti e parametri

- Gli argomenti che la funzione riceve dal chiamante devono essere memorizzati in opportune variabili *locali* alla funzione stessa dette ***parametri***
- Queste variabili locali non si definiscono nel corpo della funzione, ma nella *definizione* della funzione, specificando *di ciascuna* il *nome* e il *tipo* :

  tipo_restituito `eleva(int` **`b`**`, int` **`e`**`)`
- I *parametri* sono automaticamente inizializzati con i valori degli *argomenti*
- I parametri hanno lo stesso scope e la stessa *duration* delle variabili locali della funzione

Argomenti e parametri

- Argomenti e parametri devono *corrispondere in base alla posizione* (il primo argomento al primo parametro, etc.)
- *I nomi dei parametri sono indipendenti dai nomi delle variabili del chiamante*
- I tipi di ciascun argomento e del corrispondente parametro devono essere *compatibili:* il valore dell'argomento viene *assegnato* al parametro corrispondente con l'eventuale conversione di tipo (come una normale assegnazione `a=b;`)

Argomenti e parametri

- Se la funzione non richiedere parametri è preferibile indicare `void` tra le parentesi:

  tipo_restituito `StampaCiao(`**`void`**`)`

  Si può non indicare nulla, ma è sconsigliabile per mantenere il controllo sugli argomenti che fa il compilatore sul *prototipo* della funzione (descritto in seguito)

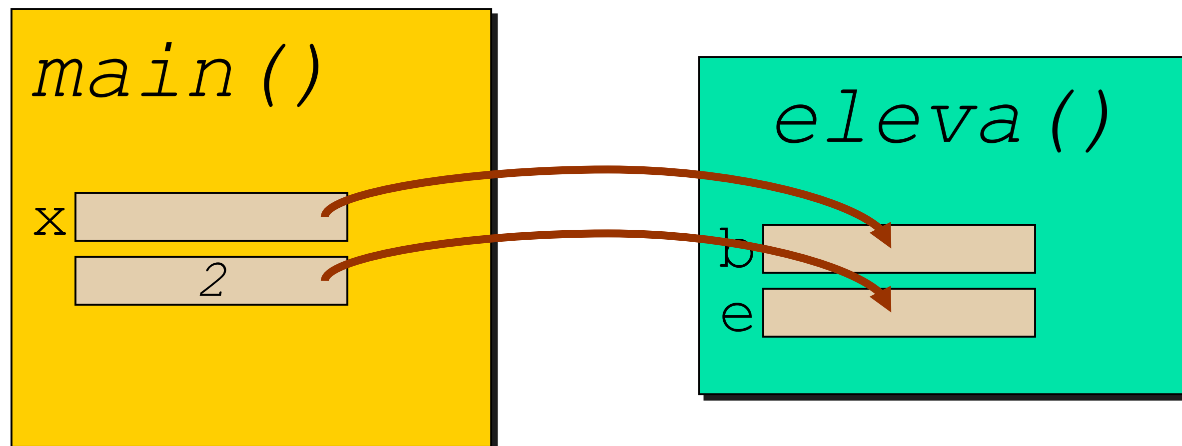
Argomenti e parametri

- **Argomenti (o parametri attuali):**
  sono i valori indicati tra le parentesi alla *chiamata* di una funzione
  `eleva(x,2)`
  possono essere variabili, costanti o espressioni, se sono espressioni prima della chiamata effettiva alla funzione tutti gli effetti collaterali sulle variabili sono portati a termine (c'è un *sequence point*)
- **Parametri (o parametri formali):**
  sono le variabili indicate tra le parentesi nella *definizione* della funzione
  `int eleva(int b, int e)`

Passaggio degli argomenti

- I dati sono passati a una funzione secondo una modalità detta ***per valore*** *(by value):*

  alla chiamata della funzione vengono *create nuove variabili* (automatiche, sullo *stack*) aventi i nomi dei parametri e in esse viene *assegnato* il valore del corrispondente argomento (idealmente si ha `b=x` e `e=2`)



Passaggio degli argomenti

- Come nelle assegnazioni, se l'argomento e il corrispondente parametro sono di tipo diverso ma *compatibili* c'è una conversione automatica al tipo del parametro (se è di tipo meno ampio può esserci un Warning)
- In memoria i parametri sono ***del tutto distinti e indipendenti*** dagli argomenti, quindi *cambiare il valore di un parametro non modifica l'argomento corrispondente,* nell'esempio visto, la modifica di `b` non si ripercuote su `x`, tantomeno la modifica di `e` può modificare la costante 2

Tipo restituito dalle funzioni

- Nella definizione

  tipo_restituito nomeFunzione (*parametri*)

  tipo_restituito indica il tipo del *valore restituito*
- Una funzione può passare (*restituire*) al chiamante **un solo valore**, questo può essere un valore numerico, un puntatore (non a una variabile *automatica* interna alla funzione poiché viene deallocata) o una `struct`; ma non può restituire un vettore (e quindi neanche una stringa)
- N.B. Non si usi mai il verbo *ritornare* al posto di *restituire*, “ritornare” in Italiano non è transitivo (lo è in Inglese e in vari dialetti italiani)

Tipo restituito dalle funzioni

- Nel caso non sia necessario che la funzione restituisca un valore (ad es. stampa solo, come l'esempio `StampaCiao`), si ha una *funzione void* (in altri linguaggi viene detta *procedura* o *subroutine*), per essa si indica il tipo `void`:

  ```
  void StampaCiao(… …)
  ```

- Se non si indica il tipo, viene supposto `int`, ma è buona norma specificarlo

Parametri e tipo del main

- Gli *argomenti* della funzione `main` possono essere:
  - `void` o nulla se non si vogliono indicare parametri sulla riga di comando (descritti in altre slide)
  - `int argc, char *argv` se si vogliono indicare parametri sulla riga di comando
- Il valore restituito dal `main`:
  - è sempre un `int`, anche se non indicato (ma è preferibile indicarlo sempre per evitare Warning)

Ritorno da una funzione

- Quando una funzione termina, l'esecuzione del programma riprende nel modulo chiamante là dove era stata sospesa
  Esempio

```
x=0.71;
y=sin(x)*2.0;
```

qui: 1) viene assegnata `x`, 2) viene calcolato `sin(x)`, 3) viene calcolato il prodotto del risultato di `sin(x)` e `2.0`, 4) viene assegnato il risultato del prodotto a `y`

Notare che l'esecuzione è sospesa *durante il calcolo dell'espressione* `sin(x)*2.0` per chiamare la funzione `sin` e riprende sostituendo a `sin(x)` il risultato della chiamata alla funzione (in una variabile temporanea)

Ritorno da una funzione

- Una funzione termina nei seguenti modi:
  - quando viene eseguita l'istruzione `return`
  - dopo aver eseguito l'ultima sua istruzione
- Una funzione può avere più istruzioni `return` (questo produce codice non strutturato)
- Il risultato prodotto da una funzione viene restituito durante il ritorno al chiamante con l'istruzione
  `return` *risultato*`;`
- Se *risultato* non ha lo stesso tipo indicato nella definizione, viene convertito automaticamente (come nelle assegnazioni, con event. Warning)

Ritorno da una funzione

- *risultato* può essere un'espressione qualsiasi:
  `return x*2+y;`
- Se il tipo restituito dalla funzione NON è `void` e la `return` è scritta senza *risultato* o non è presente al termine del corpo della funzione ci potrebbe essere un Warning (dipende dal compilatore), se il chiamante cerca di utilizzare il valore restituito si ha un comportamento indefinito
- Se il tipo restituito dalla funzione è `void`:
  - non si deve indicare *risultato* nella `return`
  - si può omettere la `return` che precede la graffa di chiusura (solo questa, non eventuali altre)

Ritorno da una funzione

- Il *valore* nella `return` del `main` viene comunicato all'interprete dei comandi del terminale che in base a questo può eventualmente eseguire altri comandi da terminale (i file contenenti comandi per il terminale sono detti *script* o *file batch* in DOS/Windows)
- Tale *valore* deve esserci, altrimenti produce un Warning e il valore restituito è indefinito

Ritorno da una funzione

- Si noti che la `return`:
  - nel `main` termina il programma (e restituisce un valore al terminale)
  - in una funzione torna al chiamante (e restituisce un valore al chiamante stesso)
- La funzione `exit` in `<stdlib.h>` termina sempre il programma, sia che sia nel `main` sia che sia in una qualsiasi altra funzione

Esempio di funzione

```
#include <stdio.h>
/* definizione della funzione */
int eleva(int b, int e)
{
   int k=1;
   while (e-- > 0)
       k *= b;
   return k;
}
```

Continua...

Esempio di funzione

Continuazione (stesso file)

```
int main()
{
   int x, y;
   printf("Introduci numero: ");
   scanf("%d", &x);
   y = eleva(x, 2);    ← chiamata funzione
   printf("%d^%d = %d\n", x,2,y);
   return 0;
}
```

- Alla chiamata, la `x` del `main` viene ***assegnata*** alla `b` di `eleva`, mentre il `2` del `main` viene ***assegnato*** alla `e` di `eleva`

Scope di una funzione

- Lo ***scope*** di una funzione indica dove essa può essere richiamata: si estende dal punto in cui viene definita fino a fine file (quindi può essere utilizzata solo dalle funzioni che nello stesso file *seguono* la sua definizione)
- Quindi una funzione per essere richiamata dal `main` *dovrebbe* essere definita prima di esso
- Ma spesso le funzioni sono definite dopo il `main` per comodità, oppure alcune funzioni si richiamano a vicenda per cui non si può stabilire chi debba essere collocata prima, altre funzioni sono definite in altri file o librerie

Scope di una funzione

- Per ovviare a questo problema la prima versione non standard del C (K&R) ha definito il concetto di *funzione implicita* come *funzione non in scope* (`f2` per `f1` nell'es. seguente).
- Tale possibilità viene mantenuta nel C89, ma questo standard introduce una modifica nella forma dei prototipi (specifica i parametri, nella versione K&R non c'erano) rendendo di fatto inutili le funzioni implicite

Scope di una funzione

```
float f1()
{
   f2();  → f1 non ha in scope f2, funz implicita
}
double f2()
{
   f1();  → f2 ha in scope f1
}
int main()
{
   f1();  → main ha in scope f1 e f2
   f2();
}
```

Scope di una funzione

- Una *funzione implicita* produce un Warning, non vengono fatti controlli sugli argomenti e il compilatore presume restituisca un `int`; nella chiamata ai parametri vengono applicate le *promozioni di default degli argomenti*, se il tipo del valore restituito non corrisponde a quello della return e/o i tipi dei parametri non sono compatibili con quelli degli argomenti il linker darà un errore
- *Promozioni di default degli argomenti:* gli argomenti di tipo intero vengono sottoposti alle promozioni integrali e gli argomenti di tipo `float` vengono convertiti in `double`

Prototipo di una funzione

- Il ***prototipo di una funzione*** è una dichiarazione (non definizione) che permette di *estendere lo scope* della funzione, quest'ultima definita "altrove"
- Ha forma sostanzialmente uguale alla riga di intestazione di una funzione, con un '`;`' al fondo
- Esempio:

```
int eleva(int b, int e);
```

Prototipo di una funzione

- I prototipi delle funzioni sono in genere tutti collocati prima del `main`, estendendo così lo scope di quelle funzioni da quel punto fino a fine file
- Se invece i prototipi sono interni a una funzione, lo scope è esteso fino alla fine della funzione
- Nel caso siano collocati tra una funzione e l'altra, lo scope è esteso da quel punto fino alla fine del file

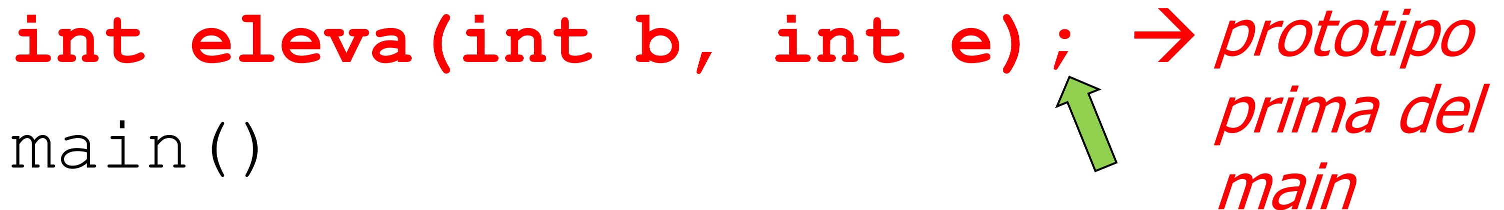
Prototipo di una funzione

- Grazie al prototipo, la *definizione* della funzione (il corpo della funzione) può essere collocata "altrove", ossia:
  - in un punto nel file successivo a dove viene chiamata (nell'esempio seguente, `eleva` è definita *dopo* il `main` dove viene utilizzata)
  - in un altro file di codice sorgente C
  - in una libreria (compilata)
- Gli *header* contengono, in particolare, i prototipi delle funzioni delle librerie C in modo che il compilatore possa verificare che siano chiamate correttamente (ad es. `<stdio.h>` contiene i prototipi di `scanf`, `printf`, etc.)

Esempio d'uso del prototipo

```
#include <stdio.h>
int eleva(int b, int e);
main()
{
   int x, y;

   printf("Introduci numero: ");
   scanf("%d", &x);
   y = eleva(x, 2);
   printf("%d^%d = %d\n", x,2,y);
   return 0;
}
```



→ *prototipo prima del main*

← *chiamata alla funzione*

Continua (stesso file)...

Esempio d'uso del prototipo

Continuazione (stesso file)

```
/* definizione della funzione */
int eleva(int b, int e)
{
    int k=1;
    while (e-- > 0)
        k *= b;
    return k;
}
```

- Si noti che la chiamata viene effettuata prima della definizione (ma dopo il prototipo)

Prototipo di una funzione

- La sintassi del prototipo di una funzione è simile alla definizione, salvo che:
  - manca il corpo
  - i *nomi* dei parametri *possono* essere omessi (ma i *tipi devono* essere presenti)
  - ha un punto e virgola alla fine

```
int eleva(int, int);
```

- I nomi dei parametri dei prototipi:
  - se non sono omessi, possono essere diversi da quelli usati nella definizione della funzione
  - sono scorrelati dagli altri identificatori (vengono di fatto ignorati, ma non da eventuali `#define`)
  - sono utili per documentare il significato dei parametri:

```
int eleva(int base, int esponente);
```

Prototipo di una funzione

- Quando viene data la sintassi d’uso di una funzione di libreria, ne viene dato il prototipo (senza ‘;’) per comprendere completamente il tipo e numero dei parametri e il tipo restituito
- Esempi
  - `FILE *fopen(const char *filename, const char *mode)`
    Indica che `fopen` richiede due parametri che non modificherà (`const`) di tipo `*char` (generico puntatore a carattere, ossia una stringa costante o variabile); il tipo restituito è un puntatore a un oggetto di tipo `FILE`
  - `char *fgets(char *s, int n, FILE *stream)`
    `fgets` richiede un parametro stringa, un intero e un file pointer, restituisce un puntatore a stringa

Prototipo di una funzione

- Esempi (segue)
  - `int fprintf(FILE *stream,`
    `const char *format, ...)`
    Indica che `fprintf` richiede come primo parametro un file pointer, come secondo una stringa (quella di formato) che non modificherà (`const`) e altri parametri facoltativi (i tre punti), inoltre restituisce un `int`
  - `int rand(void)`
    Indica che `rand` non richiede parametri (`void`) e restituisce un `int`
  - `void srand(unsigned int seed)`
    Indica che `srand` richiede un parametro di tipo `unsigned int` e non restituisce nulla (`void`)

Prototipo di una funzione

- Esempi (segue)
  - `void *memmove(void *s1, const void *s2, size_t n);`
    Indica che `memcpy` richiede due parametri di tipo puntatore a `void`, dichiara che non modificherà quanto puntato dal secondo (`const`), richiede un terzo parametro intero (`size_t` è un tipo intero), inoltre restituisce un puntatore a `void`

Esercizi

1. Si scriva un programma che per 10 volte chieda all'utente un valore e ne calcoli il logaritmo in base 2.
   Le operazioni di input e output devono essere effettuate solo dal `main`. Dopo ogni input deve essere visualizzato il risultato. Per il calcolo del logaritmo si scriva una funzione con prototipo:
   ```
   double log2(double x);
   ```
   che calcoli il valore utilizzando la formula:
   $$\log_2 x = \frac{\log_e x}{\log_e 2}$$

Esercizi

2. Si scriva un programma che per 10 volte chieda all'utente un valore intero, ne conti il numero di cifre e visualizzi il risultato. Dopo ogni input deve essere visualizzato il risultato. Per il calcolo del numero di cifre si scriva una funzione con prototipo:

```
int contacifre(int n);
```

Esercizi

3. Si scriva un programma che chieda all'utente 10 valori e di questi calcoli la radice quadrata. Per il calcolo della radice quadrata si scriva una funzione con prototipo:

```
double radice(double a, double prec);
```

che calcoli il valore approssimato a `prec` della radice quadrata di `a` con il metodo di Newton:

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{a}{x_i}\right)$$

x_i sono approssimazioni successive della radice quadrata di a. Si assuma $x_0 = a$ e si iteri fintanto che $x_i - x_{i+1} >$ `prec`.

Storage duration static

- Un oggetto con *storage duration static* rimane in memoria per tutta l’esecuzione del programma, la memoria ad esso associata viene allocata una volta sola in un segmento dati del programma e non nello stack (vedere più avanti) ed è inizializzato (una volta sola) prima dell’esecuzione del programma
- Esempi già noti sono i letterali stringa:

```
char *parola="casa";
printf("La somma vale: %d\n", sum);
```

Variabili locali static

- Sono variabili *locali* (interne a una funzione) precedute dalla keyword `static`, ad es.:
  ```
  static int cont = 0;
  ```
- `static` è uno *storage-class specifier*
- Hanno una *storage duration statica*, quindi vengono concettualmente allocate e inizializzate solo la prima volta che si entra nella funzione che le definisce (in realtà prima che il programma inizi l'esecuzione)

Variabili locali static

- Non essendo mai rimosse dalla memoria fino alla terminazione del programma, non perdono il loro valore tra una chiamata e la successiva (quando si rientra nella funzione hanno il valore che avevano all'esecuzione precedente)
- Non sono allocate nello *stack*, come invece accade per le variabili locali automatiche
- Una variabile `static` definita in una *funzione ricorsiva* è sempre la stessa in memoria ad ogni invocazione della funzione stessa

Variabili locali static

- Se non inizializzate esplicitamente, vengono automaticamente inizializzate a 0/`NULL`
- Se inizializzate esplicitamente, questo avviene (idealmente) *solo la prima volta* che si esegue la funzione
- Possono essere inizializzate dal compilatore solo con *espressioni costanti:*
  - numeri e `#define`
  - valori `enum`
  - indirizzi di memoria di variabili *statiche*
- Non possono essere inizializzate con:
  - valori `const`
  - variabili e risultati di funzioni
  - indirizzi di memoria di variabili *automatiche*

Variabili locali static

- ```
  int conta(void)
  {
     static int cont = 0;
     return ++cont;
  }
  ```

  Ogni volta che `conta` viene chiamata, essa incrementa il contatore `cont` e ne restituisce il valore, se non fosse statica ma automatica restituirebbe sempre il valore 1 perché `cont` avrebbe *duration automatica* e verrebbe inizializzata *ogni volta* a 0
  ```

Variabili locali static

- Si noti la differenza tra i seguenti casi:
  - ```
    char *caso1()
    {
        char *x = "ciao";
        return x;
    }
    ```
    Qui viene restituito un puntatore a un letterale stringa che è `static` e quindi sempre esistente
    ```

Variabili locali static

- ```
  char *caso2()
  {
        char x[100] = "ciao";
        return x;
  }
  ```
  Qui viene restituito un puntatore a una variabile *automatica* che viene rimossa dalla memoria (dallo stack) al termine dell’esecuzione della funzione (o meglio quella parte di memoria viene ridata al processo per allocarvi altro all’occorrenza, ad esempio per un’altra chiamata a funzione); teoricamente fintantoché quella parte di memoria non viene modificata il puntatore restituito punta ancora a `"ciao"`, ma è una situazione altamente inaffidabile e assolutamente da evitare; il compilatore segnala un Warning
  ```

Variabili locali static

```
char *nomeMese(int n)
{
    static char *nome[] = {
        "inesistente", "gennaio",
         "febbraio", "marzo", ecc... };
    if (n<1 || n>12)
        return nome[0];
    else
        return nome[n];
}
```

Qui viene restituito uno dei 12 puntatori a un letterale stringa (quindi `static`) come in `caso1`, quindi la keyword `static` non è necessaria, ma mettendola l'inizializzazione viene fatta una volta sola e l'esecuzione della funzione è più veloce

Passaggio per riferimento

- Il passaggio di parametri nella modalità ***per riferimento*** *(by reference)* prevede che la modifica del parametro formale modifichi il corrispondente parametro attuale (che deve essere una variabile)
- In C non esiste il passaggio per riferimento, ma lo si può *simulare* passando *per valore* alla funzione l'*indirizzo del dato* (che *deve* essere una variabile, non può essere il risultato di un calcolo perché non avrebbe un indirizzo) da far pervenire alla funzione; questo viene comunque chiamato sebbene <u>impropriamente</u> *passaggio per riferimento*

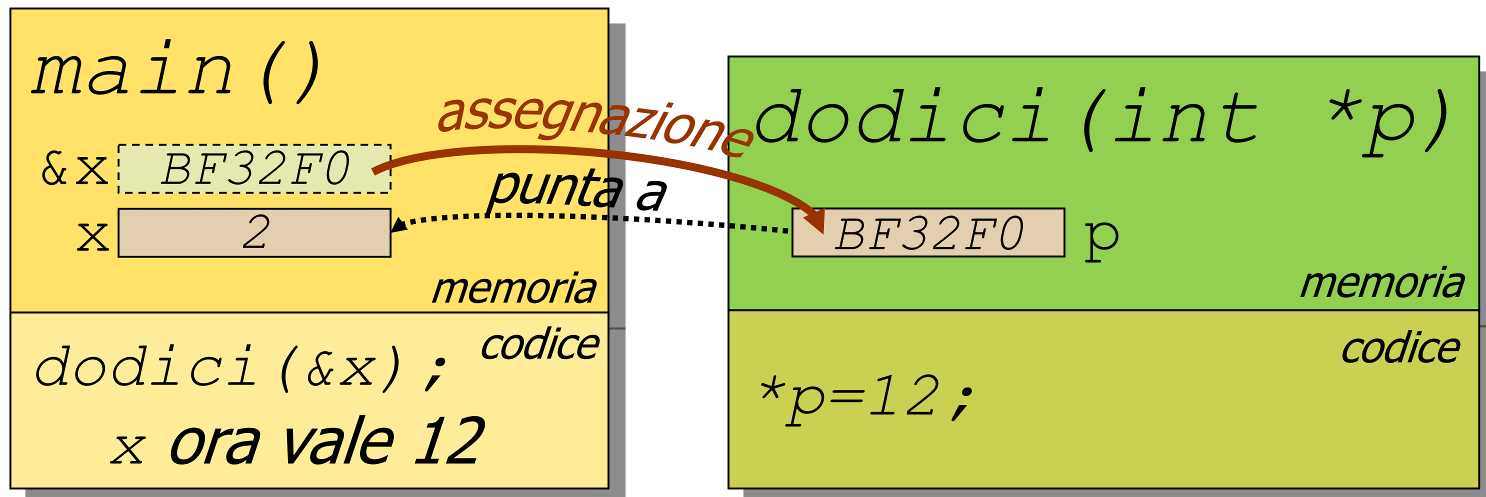
Passaggio per riferimento

- Esempio
  La funzione `dodici` assegna 12 alla variabile passata (per riferimento)

```
#include <stdio.h>
void dodici(int *);      → prototipo
int main()
{
    int x=2;
    dodici(&x);
    printf("%d\n", x);  → stampa 12
    return EXIT_SUCCESS;
}
void dodici(int *p)
{
    *p = 12;
}
```

Passaggio per riferimento

- Nell'esempio:
  - il `main` alloca `x`, le assegna 2 e chiama `dodici` passandole l'indirizzo di `x` calcolato da `&x` (quindi memorizzato in una variabile temporanea)
  - alla chiamata di `dodici`, l'indirizzo di `x` (che qui è `BF32F0`) viene assegnato a `p` (passato *per valore*)
  - `dodici` accede a `x` come `*p`, modificandola in 12
  - quando `dodici` termina, `x` vale 12



Passaggio per riferimento

- Esempio
  La funzione `swapint` scambia i valori di due variabili `int` passate (per riferimento)

```
void swapint(int *, int *); → prototipo
int main()
{
    int x=12, y=24;
    swap(&x, &y);
}
void swapint(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

Passaggio per riferimento

- Il passaggio per riferimento può essere utile anche per permettere ad una funzione di restituire più di un valore: nell'elenco degli argomenti si passano gli indirizzi di tutte le variabili a cui la funzione assegnerà risultati
- Nella `scanf` le variabili scalari sono precedute da `&` proprio perché se ne passa l'indirizzo, in quanto devono essere assegnate dalla funzione
- Esempio - Funzione a cui vengono passati 3 valori e che restituisce il valore massimo, il valore minimo e la media, il prototipo:

```
void maxMinMed(int a, int b, int c,
   int *max, int *min, double *med);
```

Passaggio per riferimento

- La funzione verrà chiamata ad es. così:

```
int aa=0, bb=0;
double cc=0.0;
…
maxMinMed(x,y,z, &aa, &bb, &cc);
```

- La funzione determinerà i valori massimo, minimo e medio e li assegnerà ai parametri:

```
*max = (x>=y&&x>=z)?x:(y>=z)?y:z;
*min = (x<=y&&x<=z)?x:(y<=z)?y:z;
*med = (x+y+z)/3.0;
```

- Alcuni compilatori danno un Warning se le variabili passate con il puntatore non sono state inizializzate o assegnate

Passaggio di vettori

- Per passare un vettore come argomento, si indica il suo nome *senza parentesi quadre:*

```
int vettore[N]={1,2,3};
float x;
x = media(vettore);
```

- Poiché il nome di un vettore è l'indirizzo di memoria del suo primo elemento, un vettore viene sempre passato per riferimento
- Il parametro formale corrispondente definirà un vettore dello stesso tipo:

```
float media(int v[])
```

Passaggio di vettori

- Nella definizione della funzione, un parametro *vettore-di-T* viene convertito in *puntatore-a-T* (ossia la forma `v[]` è convertita in `*v`), quindi le due definizioni:

  ```
  float media(int v[])
  float media(int *v)
  ```

  sono *del tutto equivalenti* (si privilegi quella che rende più comprensibile il codice)

Passaggio di vettori

- Nel caso di parametro specificato con le parentesi quadre `[]`, al loro interno in genere non si specifica nulla, se c'è una costante viene scartata
- Nel caso di parametro specificato come puntatore non c'è modo di specificare se questo punti davvero a un vettore o invece punti a un unico valore
- In entrambi i casi la mancanza di informazioni sulla dimensione permette di passare alla funzione vettori di *dimensioni diversa* (purché dello stesso tipo)

Passaggio di vettori

- La funzione deve conoscere la dimensione di un vettore passato come argomento
- Quanto eventualmente indicato tra `[]` non è utilizzabile (non è in una variabile o costante e viene quindi scartato), inoltre `sizeof` sul parametro darebbe la dimensione del puntatore, non dell'oggetto puntato
- Le possibili soluzioni sono:
  - passarla come parametro:
    `float media(int v[], int n)`
  - conoscerla a priori (es. tramite una `#define`)
  - passarla con *variabili esterne* (descritte più avanti)

Esercizi

4. Si scriva una funzione con prototipo:
   `double media(double v[], int len);`
   che calcoli e restituisca la media di un vettore di `double` passato come argomento.
   Si scriva un programma che riempia due vettori di lunghezza differente (es. `a[8]` e `b[10]`, li passi a `media` e visualizzi il risultato per ciascuno di essi. La funzione non esegua operazioni di input/output.

Esercizi

5. Si scriva un programma che chieda al'utente di inserire la dimensione e gli elementi di un vettore di `int`, inverta l'ordine degli elementi e visualizzi il vettore invertito.
   L'input del vettore avvenga con la funzione:
   ```
   int leggiVett (int vet[], int *n);
   ```
   (si noti che si passa l'indirizzo di n)
   l'inversione con la funzione:
   ```
   void inverti(int vettore[], int n);
   ```
   la visualizzazione con la funzione:
   ```
   void visualVett (int v[], int n);
   ```

Esercizi

6. Si scriva una funzione con prototipo:
   `void rovescia(char s[]);`
   che rovesci la *stringa* passata come argomento (modifica del parametro). Si scriva un programma che chieda una stringa, la passi a `rovescia` e la visualizzi.
7. Si scriva una funzione con prototipo:
   `int contastr(char a[], char x[]);`
   che conti quante volte la stringa `x` sia contenuta in `a`. N.B. "bb" in "bbb": 2 volte
8. Si scriva una funzione `undup` che modifichi una stringa eliminandone i caratteri duplicati: esempio: "ciao come_va?" → "ciao mev?"

Esercizi

9. Si scriva una funzione con prototipo:
```
void ordina(int v[], int len,
                     int ord);
```
   che ordini in senso crescente e decrescente il vettore di `int` passato come argomento.
   Il senso dell'ordinamento venga indicato dal parametro `ord` (decrescente=0, crescente=1).
   Si scriva un main di test.

Passaggio di matrici bidimens.

- Per passare una matrice bidimensionale come argomento, si indica il suo nome senza parentesi:

```
int matrice[N][M]={{1,2,3},{4,5,6}};
float x;
x = media(matrice);
```

- Il parametro formale corrispondente dichiara una matrice dello stesso tipo:

```
float media(int matrice[N][M])
```

Passaggio di matrici bidimens.

- Per le matrici bidimensionali, nel parametro formale solo la prima dimensione può non essere specificata
  ```
  float media(int matrice[][M])
  ```
- Quindi è possibile passare matrici dello stesso tipo con *diverso numero di righe*, ma tutte devono avere lo *stesso numero di colonne*
- Come per i vettori, la funzione deve comunque conoscere tutte le dimensioni della matrice

Passaggio di matrici bidimens.

- Poiché una matrice è un *vettore-di-vettori*, il suo nome è un indirizzo di memoria del tipo *puntatore-a-vettore,* NON un *puntatore-a-puntatore* in quanto il "decadimento" da vettore a puntatore può avvenire una volta sola Quindi nell'esempio: `x = media(`**`matrice`**`);` corrisponde a `x = media(&`**`matrice`**`[0]);` dove **`matrice`**`[0]` è `{1,2,3}`
- Il tipo del parametro formale corrispondente può quindi anch'esso essere definito come *puntatore-a-vettore* (di elementi di quel tipo)

Passaggio di matrici bidimens.

- Quindi le due definizioni:
  `float media(int matrice[N][M])`
  `float media(int (*matrice)[M])`
  (`N` opzionale) sono *del tutto equivalenti* e questo spiega perché non serva specificare la prima dimensione: viene persa
- Più precisamente il tipo del parametro formale `matrice` è *puntatore-a-vettore-di-M-*`int`
- Il tipo *puntatore-a-vettore-di-M-*`int` non è un *puntatore-a-puntatore-di-*`int`, quindi indicare:
  `void funz(int **matrice)`
  è sbagliato (come già detto il "decadimento" da vettore a puntatore avviene una volta sola)

Passaggio di matrici bidimens.

- Data la definizione: `int matrice[N][M];` l'indirizzo di memoria dell'elemento `matrice[`*i*`][`*j*`]` viene calcolato con:
  `indirizzo_di_matrice + M*`*i* + *j*
  Come si vede il numero delle righe (`N`) non serve, ma quello delle colonne (`M`) sì
- Questo potrebbe suggerire di usare una matrice come fosse un lungo vettore, in effetti questo è in genere possibile, ma lo Standard non accetta che si acceda agli elementi oltre la fine di una riga (potrebbe comunque essere eventualmente permesso dal compilatore)

Passaggio di matrici bidimens.

- Per passare a una funzione una matrice con qualsiasi numero di righe e colonne si rimanda alle slide sull'allocazione dinamica

Passaggio di matrici VLA

- Si può passare a una funzione una matrice VLA come una qualsiasi matrice a dimens. costanti
- Anche per le matrici VLA usate come parametri è possibile esplicitare il legame tra la matrice e le sue dimensioni indicandole prima della matrice e tra le parentesi del vettore
  Ad esempio la *definizione* della funzione:
  ```
  int somma(int n, int m, int matrice[n][m])
  ```
  per la quale *prototipi* equivalenti sono:
  ```
  int somma(int n, int m, int matrice[n][m]);
  int somma(int n, int m, int matrice[*][*]);
  int somma(int n, int m, int matrice[][*]);
  int somma(int n, int m, int matrice[][m]);
  ```

Esercizi

10. Si scriva una funzione con prototipo:
```
double media(double M[][10],
                  int righe, int colonne);
```
che calcoli e restituisca la media di una matrice di qualsiasi numero di righe ed esattamente 10 colonne passata come argomento. Si scriva un programma di test che riempia due matrici di dimensioni 8x10 e 12x10, le passi a `media` e visualizzi il risultato per ciascuna di esse.
La funzione non faccia input/output.
N.B. passare il numero di colonne è superfluo.

Passaggio di vettori multidim.

- Quanto visto per le matrici bidimensionali può essere esteso ai vettori con qualsiasi numero di dimensioni
- In particolare:
  - nel parametro formale si può tralasciare la dimensione del solo primo parametro
  - il parametro formale è un puntatore a un vettore di *n* vettori di *m* vettori di *p* vettori (ecc.) di tipo T
  - la funzione deve conoscere i valori di tutte le dimensioni (*n*, *m*, *p*, ecc.)
  - possono essere passati vettori multidimensionali di elementi di tipo T in cui solo la prima può avere valori diversi (a meno che non siano matrici VLA)

Variabili esterne

- Un file C è composto da funzioni, prima della prima funzione, tra una funzione e l'altra e dopo l'ultima funzione si è all'*esterno* delle funzioni
- Se un progetto è composto da più file di funzioni, per ogni file tutto ciò che è scritto negli altri file è esterno alle funzioni di quel file
- Le *variabili esterne* (o *globali*) sono <u>definite</u> (riserva memoria) *esternamente* alle funzioni:
  - in testa al file, tipicamente dopo le direttive `#include` e `#define`
  - oppure tra una funzione e un'altra
  - oppure in altri file

Variabili esterne

- Una variabile esterna definita in un file è visibile e utilizzabile:
  - in tutte le funzioni che, *nello stesso file*, seguono la definizione
  - in altri file dello stesso progetto (o meglio che saranno collegati insieme dal linker) se in questi si definisce l'equivalenza (condivisione della porzione di memoria) con questa variabile (mediante `extern`)
- Una variabile locale (interna a una funzione) con lo stesso nome di una esterna "nasconde" (a partire dalla riga dove è definita) quella esterna alla quale quindi non si può più accedere: si evitino queste evenienze

Variabili esterne

- Nelle funzioni che *nello stesso file* seguono la definizione di una variabile esterna, non serve indicare nulla per accedere a quella variabile
- Ma se si desidera, per maggior chiarezza è possibile "ridefinirla" anteponendo la parola chiave `extern` (in realtà questa è una *dichiarazione* perché non alloca memoria, essendo già stata allocata dalla *definizione*)
- Nelle funzioni che *nello stesso file* precedono la definizione di una variabile esterna e in quelle di *altri file* del progetto è *necessario* dichiararla con `extern` prima del suo utilizzo

Variabili esterne

```
int uno;
long due;
main()
{
   extern int uno;   → questa riga non serve!
   long due;   →  non è la variabile esterna due!
   extern int tre;  → questa riga serve!
   uno = 12;
   due = 4L;
   tre = 27;
}
int tre;
```

sono la stessa (int uno; ↔ extern int uno;)

sono la stessa, ma è definita dopo (extern int tre; ↔ int tre;)

Variabili esterne

- Le variabili esterne e le funzioni hanno *scope di file:* ogni funzione vede e può richiamare tutte le funzioni presenti suo file e tutte le variabili esterne del suo file (con le limitazioni di scope e le soluzioni già indicate)

Variabili esterne

- Hanno *storage duration statica:* mai rimosse dalla memoria, inizializzate automaticamente a 0/`NULL` salvo inizializzazione esplicita
- Sono spesso usate per non dover passare molte variabili come argomenti alle funzioni
- Possono rendere più veloce la chiamata a funzione in quanto passano meno valori
- Ma rendono poco evidente il flusso dei dati nel programma, causando spesso errori difficili da scovare, inoltre riducono la riutilizzabilità del codice: *si usino il meno possibile*

extern

- Ricapitolando: `extern` *dichiara* (ossia non alloca memoria) una variabile, indicando al compilatore che è *definita* (quindi con allocazione di memoria) altrove:
  - altrove nello stesso file
  - in un altro file oggetto, anche una libreria
- `extern` è uno *storage-class specifier*
- In fase di linking tutte le dichiarazioni verranno ricondotte all'unica definizione
- Per i vettori non serve indicare nulla tra le parentesi quadre

extern

- Quando si dichiara un vettore `extern`, si deve indicare la forma sintattica di un vettore e non di un puntatore: il decadimento del nome di un vettore a puntatore si ha solo nelle espressioni
- Esempio

```
void funzione()
{
   extern int v[]; → dichiarazione interna corretta
   extern int *w;  → dichiarazione interna errata
}
int v[N];      → definizione esterna
int w[M];      → definizione esterna
```

extern

- Una variabile con lo specificatore `extern` (dovunque sia stata *definita*, nello stesso file o in un altro) ha scope che va dal punto stesso dove è *dichiarata* fino:
  - *a fine file* se collocata esternamente alle funzioni, anche se è definita in altro file continua ad essere esterna
  - *a fine blocco* se collocata internamente a un blocco di codice (una funzione o un blocco tra graffe), quindi non è esterna per quel file che contiene la dichiarazione

extern

```
#include<…>
extern int a;
main()
{
   a = 12;
}
void fun1()
{
   extern int b;
   a = 21; b=55;
}
int c;
int fun2()
{
   return a + c;
}
int b;
```

Scope di a, definita in altro file

Scope di b, definita sotto

Scope di c, definita qui

a

b

c

Parametri const

- Il modificatore `const` applicato ai parametri formali impedisce che all'interno della funzione si possa modificarne il valore

```
int funzione(const int v)
```

- Permette di proteggere i parametri da una successiva inavveduta modifica (per prevenire errori di programmazione)
- `const` è utile anche come documentazione: il prototipo stesso della funzione (ad es. di libreria) permette di sapere che il parametro non verrà modificato
- `const` e `int` possono essere invertiti di posto

Esercizi

11. Si scriva la funzione `sommaVett` che calcoli la somma di due vettori. Si scriva un `main` che chieda la dimensione dei vettori (max 100), ne chieda i valori, li passi alla funzione e visualizzi il vettore dei risultati. *Si protegga il contenuto dei vettori dalla modifica*. La funzione non faccia input/output.
    Per non usare variabili esterne o variabili locali `static`, bisogna passare alla funzione anche il vettore dei risultati.

Parametri puntatori const

- Nel caso più frequente si ha una definzione di funzione come la seguente:
  `int funzione(`**`const`**` int *v)`

  Con questa sintassi si specifica che l'*oggetto puntato* da `v` non verrà modificato dalla funzione
- La keyword `const` può essere collocata in diversi modi, i 4 possibili casi sono dettagliati nelle slide seguenti

Parametri puntatori const

1. **Puntatore variabile** a **dati variabili**

```
int f(int *p)
{
   *p = 12;   → OK, dato variabile
   p++;       → OK, puntatore variabile
}
```

- Si può passare un *`int*`*
- **Non si può passare** un *`const int*`* perché dentro la funzione nulla vieterebbe di poter cambiare il valore alla variabile puntata:

```
int x=12;
const int *y=&x;
f(y);   → ERRORE
```

Parametri puntatori const

2. **Puntatore variabile** a **dati costanti**

```
int f(const int *p) /* int const */
{
   int *q;   /* non e' const */
   const int *y;
   *p = 12;  → NO, dato costante
   p++;      → OK, puntatore variabile
   q=p;      → NO, q potrebbe modificare *p
   y=p;      → OK, y non può modificare *p
}
```

- Si può passare un `int*` (c'è conversione di tipo automat. perché si passa a un tipo più restrittivo)
- Si può passare un *`const int*`*

Parametri puntatori const

2. *Seguito* **(Puntatore variabile** a **dati costanti)**

Note:

- Si utilizza tipicamente per passare a una funzione un puntatore ad un oggetto di tipo aggregato (una `struct` o un vettore) richiedendo al compilatore di verificare che l’oggetto puntato non sia modificato
- Per passare come argomento un puntatore di tipo `Tipo**` dove è richiesto come parametro un puntatore di tipo `const Tipo**` è necessario un cast:

```
void funz(const int **x);
int a, *p=&a, **q=&p;
funz( (const int **)q);
```

Parametri puntatori const

3. **Puntatore costante** a **dati variabili**

```
int f(int * const p)
{*p = 12;   → OK, dato variabile
   p++;}    → NO, puntatore costante
```

- Si può passare un `int*`
- Si può passare un `const int*`

4. **Puntatore costante** a **dati costanti**

```
int f(const int * const p)
{*p = 12;   → NO, dato costante
   p++;}    → NO, puntatore costante
```

- Si può passare un `int*` (c'è conversione di tipo automat. perché si passa ad un tipo più restrittivo)

Esercizi

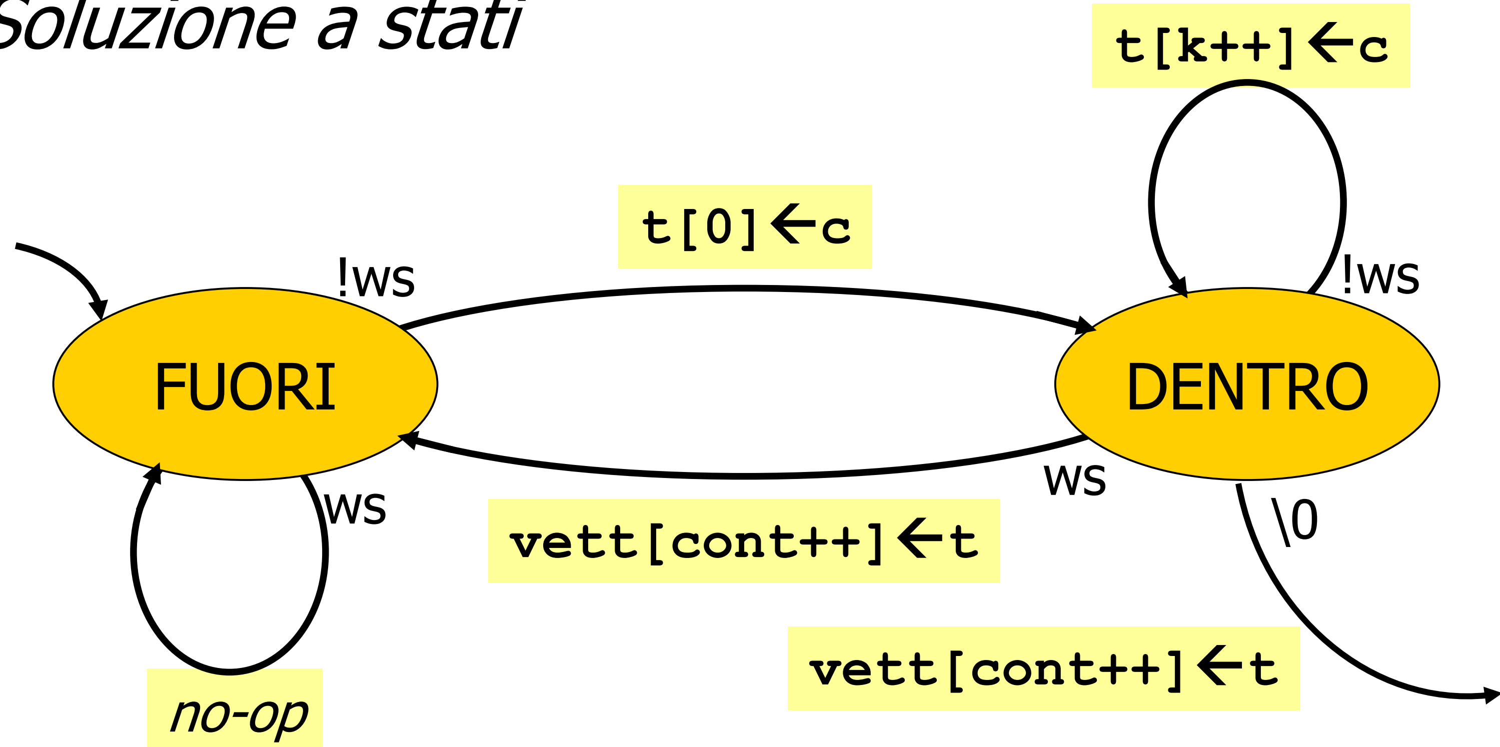
12. Un file di testo denominato `Parole.txt` contiene una lista di parole, una per riga. Non è noto a priori di quante righe sia composto. Si scriva un programma che chieda all'utente di introdurre una parola e visualizzi tutte le parole presenti nel file che anagrammate danno la parola introdotta. Si scrivano due funzioni: una che riordina alfabeticamente le lettere di una stringa ("telefono" → "`eeflnoot`") e un'altra che modifichi una stringa trasformandone tutti i caratteri in minuscolo.

Esercizi

13. Scrivere una funzione con prototipo
```
int parseToIntVect(char *stringa,
                   int vett[], int n);
```
che analizzi la `stringa` data estraendo da essa i primi `n` *valori numerici interi* (non singole cifre) e li inserisca nel vettore `vett[]`. Se vi sono più di `n` valori, i successivi vengano ignorati. La funzione restituisca il numero di elementi letti (che possono quindi essere meno di `n`). Si supponga che il file non contenga caratteri diversi da cifre e white spaces. Si scriva un main di test.

Esercizi

Soluzione a stati



ws=white space
`t`=stringa, contiene i char da trasformare in numero
`cont`=contatore valori
`c`=carattere i-esimo della stringa (`stringa[i]`)

Esercizi

14. Si scriva un programma che permetta di calcolare il prodotto di 2 numeri interi senza segno introdotti dall'utente e composti ciascuno da un massimo di 1000 cifre. Modularizzare il codice utilizzando opportune funzioni di supporto per:
    - moltiplicare un numero per una cifra
    - fare lo shift di n posizioni a sinistra di un numero
    - sommare due numeri

Documentazione delle funzioni

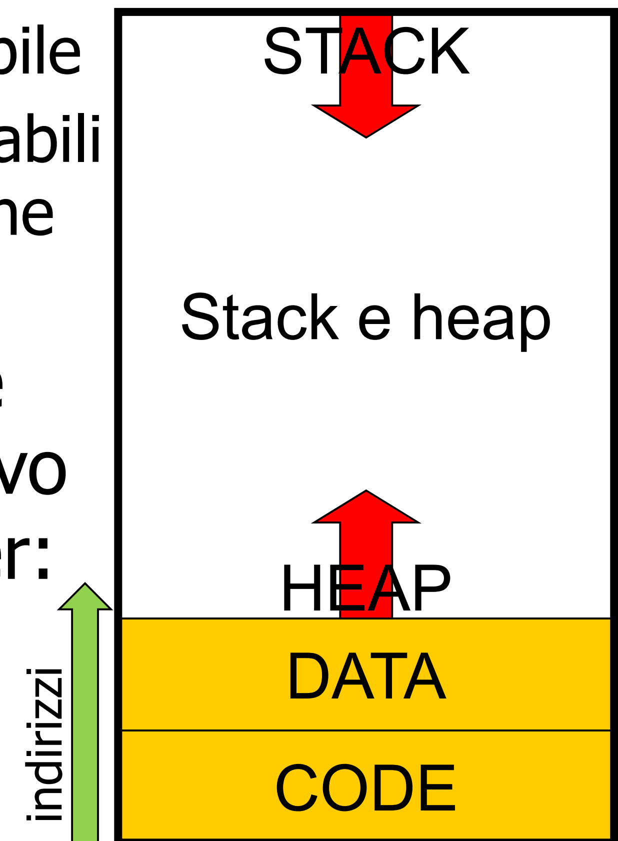
- In testa a ogni funzione è utile inserire un commento con informazioni su di essa, per documentazione e perché esistono programmi in grado di estrarle e formattarle in una pagina HTML (es. Doxygen), in particolare:
  - *Nome e descrizione* (a che cosa serve la funzione)
  - *Parametri* (tipo e descrizione di ciascuno)
  - *Valore restituito* (tipo e descrizione)
  - *Effetti secondari* (es. modifica di variabili esterne)
- La descrizione dei parametri può contenere anche indicazioni sui valori accettabili (dette *pre-condizioni*, es. "$x \geq 0$"); se queste sono soddisfatte, vengono garantite delle "post-condizioni" (es. "risultato ≥ 0")

Documentazione delle funzioni

- La funzione è libera di controllare se le pre-condizioni sono rispettate o no e agire eventualmente in maniera opportuna
- Per il controllo si possono utilizzare istruzioni `if` o `assert` (descritte più avanti)
- Spesso i controlli si tralasciano per non ridurre la velocità del programma stesso, ma il programma è meno sicuro, in altri casi è il sistema al run-time che si accorge del problema (es. $x<0$ in una `sqrt`) e blocca il programma

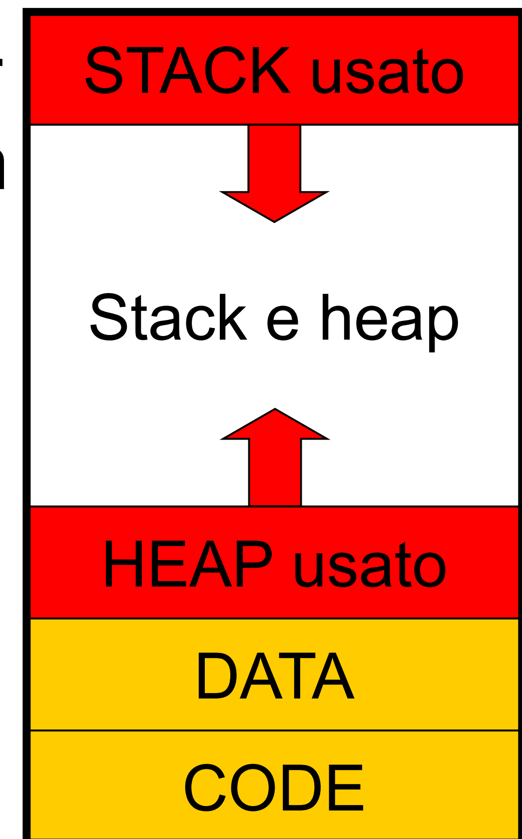
Chiamata di funzione - dettagli

- Il programma compilato è costituito da due parti distinte:
  - *code segment:* codice eseguibile
  - *data segment:* costanti e variabili note alla compilazione (statiche ed esterne)
- Quando il programma viene eseguito, il Sistema Operativo alloca spazio di memoria per:
  - il code segment (CS)
  - il data segment (DS)
  - lo *stack* e lo *heap* (condivisi)



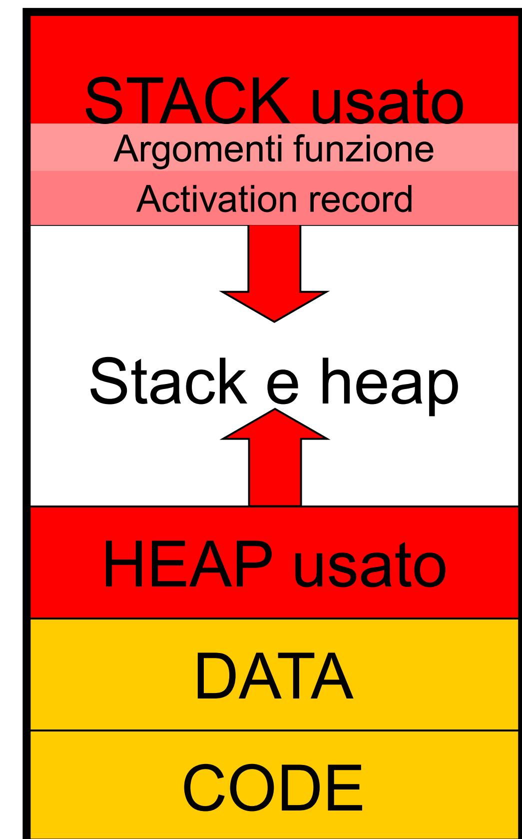
Chiamata di funzione - dettagli

- Lo *stack* (“pila”) contiene inizialmente le variabili *locali* della funzione `main()`
- Lo *heap* (“mucchio”) serve per contenere i blocchi di memoria allocati dinamicamente con funzioni `malloc()` (in altre slide), inizialmente è vuoto
- Stack e heap occupano la stessa area di memoria, ma crescono nel senso indicato dalle frecce, lo spazio può esaurirsi e il programma malfunzionare



Chiamata di funzione - dettagli

- Quando viene chiamata una funzione, lo stack aumenta di dimensioni: vengono prima copiati i valori dei suoi argomenti e poi vi viene allocato un ***Activation Record*** (anche detto ***stack frame***) in cui sono allocate le variabili locali automatiche della funzione e altro
- Quando la funzione termina, l'AR e gli argomenti vengono rimossi dallo stack che quindi ritorna nello stato che aveva prima della chiamata alla funzione



Chiamata di funzione - dettagli

- Nell'Activation Record viene anche memorizzato l'***indirizzo di ritorno*** dalla funzione: l'indirizzo di memoria che contiene l'istruzione del modulo chiamante da cui continuare l'esecuzione dopo che la funzione è terminata
- Queste operazioni di allocazione e deallocazione di spazio sullo stack e in generale il meccanismo di chiamata e ritorno da una funzione *richiedono tempo*
- In caso di necessità di *elevate* performance, si può cercare di limitare il numero delle chiamate a funzione, a costo di ricopiare lo stesso codice in più punti (eventualmente anche con *macro con argomenti*)