




**UNIVERSITÀ DEGLI STUDI  
DEL SANNIO** Benevento  
**DING**  
DIPARTIMENTO DI INGEGNERIA

**CORSO DI "PROGRAMMAZIONE I"**

Prof. Franco FRATTOLILLO  
Dipartimento di Ingegneria  
Università degli Studi del Sannio


Franco FRATTOLILLO - Dipartimento di Ingegneria - Università degli Studi del Sannio    Corso di "Programmazione I" - Corso di Laurea in Ingegneria Informatica / ExAT    1



**Dichiarazione e definizione di una variabile**

- Esiste una differenza tra *dichiarazione* e *definizione* di una variabile:
  - con la dichiarazione si rende noto al compilatore le proprietà di una variabile (tipo), ma non si alloca spazio in memoria
  - con la definizione si crea spazio in memoria riservata alla variabile


Franco FRATTOLILLO - Dipartimento di Ingegneria - Università degli Studi del Sannio    Corso di "Programmazione I" - Corso di Laurea in Ingegneria Informatica / ExAT    2



**Dichiarazione e definizione di una funzione**

- Esiste una differenza tra *definizione* e *dichiarazione* di una funzione:
  - con la definizione si definisce la funzione, cioè si riporta il corpo della funzione racchiuso fra parentesi graffe
  - con la dichiarazione si rende visibile la funzione attraverso l'indicazione del suo prototipo, cioè si rende possibile invocarla e si specifica come invocarla
    - la lista dei parametri può essere muta

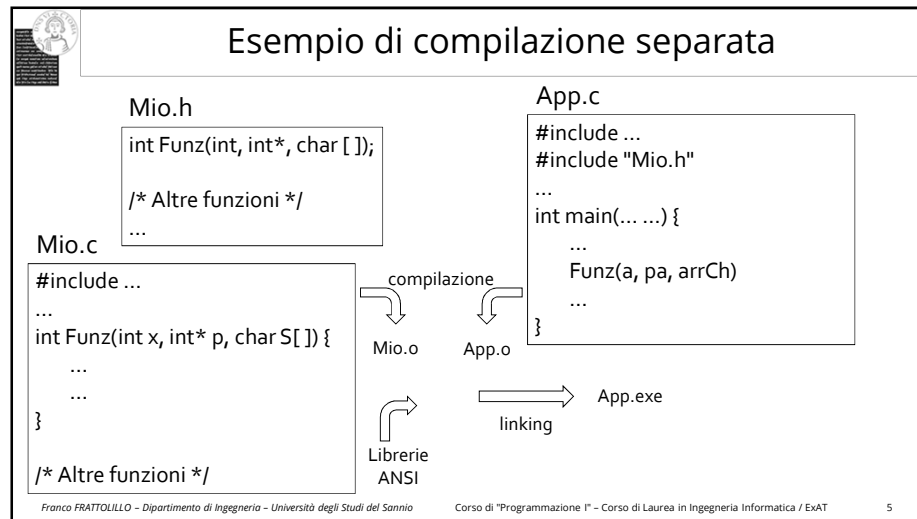
Franco FRATTOLILLO - Dipartimento di Ingegneria - Università degli Studi del Sannio    Corso di "Programmazione I" - Corso di Laurea in Ingegneria Informatica / ExAT    3



**Compilazione separata**

- Un programma può essere scomposto in diversi file, ognuno dei quali implementa un sottoinsieme di funzioni
- Ogni file può essere a sua volta suddiviso in due parti:
  - Un file *header*, tipicamente con estensione .h, contenente i prototipi delle funzioni e dichiarazioni di variabili che devono essere esportati
  - Un file .c contenente l'implementazione delle funzioni
- I file di implementazione (.c) devono includere i file header perché il compilatore possa associare i prototipi alle definizioni delle funzioni e delle variabili

Franco FRATTOLILLO - Dipartimento di Ingegneria - Università degli Studi del Sannio    Corso di "Programmazione I" - Corso di Laurea in Ingegneria Informatica / ExAT    4



### Classi di memorizzazione

- Esistono 4 diversi specificatori di classe di memoria per le variabili, rappresentati dalle label:
  - extern
  - auto
  - static
  - register // obsoleta
- A ciascuno di esse corrisponde un diverso tempo di vita della variabile ed un diverso uso

Franco FRATTOLILLO - Dipartimento di Ingegneria - Università degli Studi del Sannio Corso di "Programmazione I" - Corso di Laurea in Ingegneria Informatica / ExAT 6

### Scope e classe di memoria di una variabile

- Lo "scope" di una variabile è il range di visibilità di una variabile, ossia l'insieme dei moduli in cui la variabile risulta essere visibile
- La "classe di memoria" di una variabile definisce il tempo di vita della variabile, cioè il tempo durante il quale la variabile esiste durante l'esecuzione del programma

Franco FRATTOLILLO - Dipartimento di Ingegneria - Università degli Studi del Sannio Corso di "Programmazione I" - Corso di Laurea in Ingegneria Informatica / ExAT 7

### Variabili globali

- Le variabili globali sono definite esternamente a tutte le funzioni e sono nella visibilità dei file, a partire dal punto di definizione
- Hanno un tempo di vita pari a quello dell'intero programma e conservano il proprio valore durante l'intera esecuzione del programma
- È possibile rendere visibile una variabile globale definita in un file in un qualsiasi altro file del programma mediante una definizione extern

Franco FRATTOLILLO - Dipartimento di Ingegneria - Università degli Studi del Sannio Corso di "Programmazione I" - Corso di Laurea in Ingegneria Informatica / ExAT 8

## Variabili e funzioni esterne

- Una variabile viene specificata "esterna" quando la sua definizione è posta al di fuori del file che la utilizza
- Una variabile è esterna quando:
  - è definita al di fuori delle funzioni, cioè globalmente;
  - definita nel file
  - è dichiarata tale esplicitamente attraverso la label extern; in tal caso la sua definizione può anche essere successiva al modulo che la utilizza
- Una funzione può sempre essere dichiarata esterna, in quanto il C non permette di definire funzioni annidate
- Le variabili e le funzioni esterne vengono linkate esternamente
  - è possibile riferire tali variabili e funzioni anche da funzioni presenti in file compilati separatamente

## Scope di una variabile esterna

- Va dal punto della sua dichiarazione fino alla fine del file in cui è dichiarata
- Comprende sia il file in cui è presente la sua definizione, che tutti i file diversi da quello sorgente in cui è presente una sua dichiarazione attraverso la label extern
  - tali variabili mantengono il valore memorizzato anche quando le funzioni che le hanno utilizzate terminano
  - possono essere usate per scambiare dati tra funzioni

## Esempio di variabili globali

Mio.c

```
#include ...
...
extern int x;

int Funz(int k, int* p, char S[]) {
    ...
    x=...
    ...
}

/* Altre funzioni */
```

App.c

```
#include ...

int x=10;
...
int main(... ..) {
    ...
    k = x+...
}

int y=20;
...
```

## Esempio

Primo.c

```
int numero;
extern int incrementa();

main() {
    numero = 0;
    incrementa();
}
```

Secondo.c

```
extern int numero;
int incrementa() {
    numero++;
}
```

## Variabili automatiche

- Una variabile definita all'interno di un blocco di funzione (variabile locale) o anche preceduta dal prefisso auto viene considerata automatica
- Di default ogni variabile definita all'interno di una funzione o blocco è considerata automatica, per cui il prefisso auto è opzionale
- Una variabile automatica è referenziabile solo all'interno della funzione o del blocco in cui è stata definita
- Una variabile auto o locale è deallocata quando si esce dalla funzione o dal blocco in cui è definita

## Esempio di variabili automatiche

```
#include ...
...
int main(... ..) {
    int x;
    ...
    for(...) {
        int k;
        ...
    }
}
```

25/03/2022

Persistenze e private

## Variabili e funzioni static, contatori

- Una variabile dichiarata static mantiene il suo valore per tutta la vita del programma (persistente)
  - static int x;
- Lo scope di una variabile static definita all'interno di una funzione o blocco è uguale allo scope di una variabile automatica
- Lo scope di una variabile static definita in un file è tutto il file nel quale viene definita (privata del file)
- Lo scope di una funzione static è uguale a quello di una variabile static definita in un file, per cui risulta invocabile solo all'interno del file in cui è definita (privata del file)

## Esempio di variabile static

```
#include ...
...
void incrementa(void) {
    int x = 0;
    ++x;
    printf("%d\n", x);
}

int main(...) {
    incrementa();
    incrementa();
    incrementa();
}
```

La x viene  
inizializzata  
una sola volta  
all'atto della  
compilazione

```
#include ...
...
void incrementa(void) {
    static int x = 0;
    ++x;
    printf("%d", x);
}

int main(...) {
    incrementa();
    incrementa();
    incrementa();
}
```

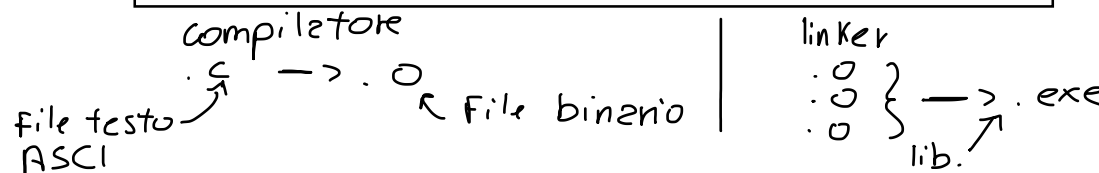
mantiene il  
suo valore

## Inizializzazione

- Se non **inizializzate** esplicitamente, le variabili **globali** e **statiche** sono **inizializzate a zero**
  - altrimenti possono essere inizializzate con valori costanti

```
int x=1;
int y=x+1; /* errato */
char c = 'b';
int vect[ ] = {12, 13, 34, 134};
```
- Le variabili **locali** (automatiche) **devono essere sempre inizializzate prima del loro utilizzo** altrimenti **assumeranno un valore indefinito**
  - è buona norma, comunque, **inizializzare qualsiasi variabile prima di utilizzarla**

Franco FRATTOLILLO - Dipartimento di Ingegneria - Università degli Studi del Sannio    Corso di "Programmazione I" - Corso di Laurea in Ingegneria Informatica / ExAT    17



## Qualificatore const

- Può precedere la definizione di variabili in cui c'è **inizializzazione** ed **impedisce la modifica del valore**
  - `const double pigreco = 3.14;`
  - `pigreco = 10;` /\* errore \*/
- Le variabili **const** sono **diverse dalle costanti definite con la direttiva #define del preprocessore**
  - quest'ultime sono semplicemente dei nomi simbolici associati a stringhe di caratteri, e il preprocessore sostituisce nel codice ogni occorrenza del nome simbolico con la stringa corrispondente

Franco FRATTOLILLO - Dipartimento di Ingegneria - Università degli Studi del Sannio    Corso di "Programmazione I" - Corso di Laurea in Ingegneria Informatica / ExAT    18

preprocessore: elaborazioni testo - testo  
 .c → .c + direttive el preprocessore

## Il preprocessore C

- Il **preprocessore C** è un tool del linguaggio che **fornisce le seguenti possibilità**
  - definizione di costanti e di macro**
  - inclusione di file**
  - compilazione condizionale**
- I **comandi** (o direttive) del preprocessore **iniziano con #** nella prima colonna del file sorgente e **non richiedono il ';' alla fine della linea**
- Il **preprocessore agisce prima della compilazione vera e propria**, effettuando una serie di sostituzioni **testuali nel file sorgente**

Franco FRATTOLILLO - Dipartimento di Ingegneria - Università degli Studi del Sannio    Corso di "Programmazione I" - Corso di Laurea in Ingegneria Informatica / ExAT    19

## Preprocessore C: costanti

- Attraverso la direttiva **#define del preprocessore è possibile specificare delle costanti**:

```
#define nome testo da sostituire
#define MAXLEN 100
#define YES 1
#define NO 0
#define ERROR "File non trovato\n"
```
- È **uso indicare per le macro di sostituzione le lettere maiuscole**
- Perché usare le macro di sostituzione?
  - favoriscono la leggibilità del programma
  - consentono un facile riuso del codice

Franco FRATTOLILLO - Dipartimento di Ingegneria - Università degli Studi del Sannio    Corso di "Programmazione I" - Corso di Laurea in Ingegneria Informatica / ExAT    20

## Preprocessore C: macro

- L'uso della direttiva `#define` consente anche di definire delle macro
- Una macro è una porzione di codice molto breve che è possibile rappresentare attraverso un nome
  - il preprocessore provvede ad espandere il corrispondente codice in linea
- Una macro può accettare degli argomenti, nel senso che il testo da sostituire dipende dai parametri utilizzati all'atto della chiamata
  - il preprocessore espande il corrispondente codice in linea, avendo cura di rimpiazzare ogni occorrenza del parametro formale con il corrispondente argomento reale

Franco FRATTOLILLO - Dipartimento di Ingegneria - Università degli Studi del Sannio    Corso di "Programmazione I" - Corso di Laurea in Ingegneria Informatica / ExAT    21

## Macro: Esempi

```
#define square(x) ((x)*(x))
#define MIN(a,b) ((a<b)? (a) : (b) )
#define ASSERT(expr) if (!(expr)) printf("error")
```

- Le linee del file sorgente
 

```
square(2);
MIN(2,3);
ASSERT(a > b);
```
- sono sostituite con
 

```
((2)*(2));
( 2 < 3 ) ? (2) : (3) );
if (!(a > b)) printf("error");
```

Franco FRATTOLILLO - Dipartimento di Ingegneria - Università degli Studi del Sannio    Corso di "Programmazione I" - Corso di Laurea in Ingegneria Informatica / ExAT    22

## Cosa non è una macro

- Le macro NON sono funzioni
  - per esempio, sugli argomenti delle macro non esiste controllo sui tipi
- Una chiamata del tipo `MIN(i++,j++)` è sostituita con `((i++ > j++) ? (i++) : (j++) )`;
- È importante stare attenti all'uso delle parentesi; ad esempio in: `#define square(x) x * x`
- Una chiamata del tipo `x = square(3+1);` genera `x = 3+1 * 3+1; --> x = 7`

Franco FRATTOLILLO - Dipartimento di Ingegneria - Università degli Studi del Sannio    Corso di "Programmazione I" - Corso di Laurea in Ingegneria Informatica / ExAT    23

## Preprocessore C: annullare una definizione

- Per annullare una definizione basta usare `#undef name`
- Ad esempio, la funzione `getchar( )` è una macro definita nella libreria standard `stdio.h`
  - se s'intende ridefinire questa macro, occorre annullare prima la sua definizione `#undef getchar( )`

Franco FRATTOLILLO - Dipartimento di Ingegneria - Università degli Studi del Sannio    Corso di "Programmazione I" - Corso di Laurea in Ingegneria Informatica / ExAT    24

## Preprocessore C: inclusione di file

- Il comando di inclusione `#include` permette di inserire il file specificato nel file sorgente che contiene la direttiva di inclusione, a partire dal punto in cui è presente la direttiva

→ `#include "const.h"`

→ `#include <stdio.h>`

- nel primo caso il file da includere verrà ricercato nella directory corrente
- nel secondo verrà cercato in quella di default, definita all'atto dell'installazione del compilatore C

## Preprocessore C: compilazione condizionale

*multipiattaforma*

- Le direttive :  
`#if #ifdef #ifndef #elif #else #endif`
- consentono di subordinare la compilazione di alcune parti di codice alla valutazione di alcune costanti in fase di compilazione  

```
#if <espressione_costante>
    <statement_1>
#else
    <statement_2>
#endif
```
- Se l'espressione costante specificata, valutata in compilazione, ritorna TRUE, allora verranno compilati gli `statement_1`; altrimenti verranno compilati gli `statement_2`

## Preprocessore C: compilazione condizionale

- Queste direttive vengono spesso usate per impedire che una certa porzione di codice possa essere inclusa più di una volta
- Ad esempio, se abbiamo diversi file sorgenti, ciascuno dei quali utilizza delle funzioni presenti in un file `header.h`, per evitare che ciascun file includa l'intero file, si può condizionare l'inclusione alla condizione che un certo simbolo sia definito oppure no

## Compilazione condizionale

- `#if !defined(HEADER)`
- `#define HEADER`
- ... contenuto di `header.h`
- `#endif`
- L'espressione costante `defined` assume valore 1 se il nome racchiuso tra parentesi è stato già definito, 0 altrimenti
  - in questo modo il primo file che presenta la direttiva `#include "header.h"`, valuta l'espressione `#if`, trova che `HEADER` non è stato definito, lo definisce ed include il codice compreso tra `#if` e `#endif`
  - i file successivi, all'atto dell'inclusione, valutano l'espressione, trovano che `HEADER` è stato già definito e saltano alla riga successiva di `#endif`, non includendo il file `header.h`