Programmazione B
Ingegneria e Scienze Informatiche - Cesena
A.A. 2021-2022

Visibilità e tempo di vita delle variabili

Catia Prandi - catia.prandi2@unibo.it

Credit: Pietro Di Lena

Q: What is the best naming prefix for a global variable?
A: //

Caratteristiche di una variabile

- Abbiamo già visto le seguenti caratteristiche di una variabile:
  - **Nome**: definito come identificatore.
  - **Tipo**: specifica la classe di valori che la variabile può assumere e gli operatori applicabili su tale tipo di dato.

Caratteristiche di una variabile

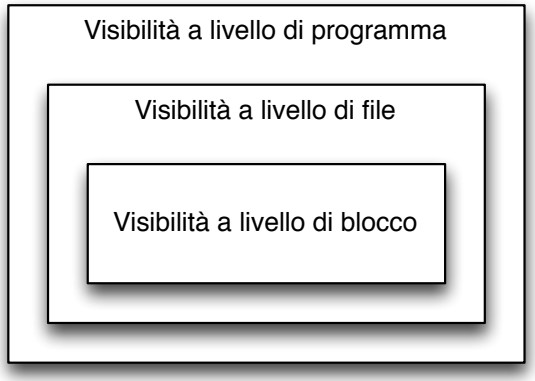
- Abbiamo già visto le seguenti caratteristiche di una variabile:
  - **Nome**: definito come identificatore.
  - **Tipo**: specifica la classe di valori che la variabile può assumere e gli operatori applicabili su tale tipo di dato.
- Vedremo adesso principalmente le seguenti caratteristiche:
  - **Visibilità** (o **scope**): porzione di programma in cui la variabile è nota e può quindi essere utilizzata.
  - **Tempo di vita**: intervallo di tempo in cui resta valida l'associazione tra nome della variabile e indirizzo fisico di memoria.

Caratteristiche di una variabile

- Abbiamo già visto le seguenti caratteristiche di una variabile:
  - **Nome**: definito come identificatore.
  - **Tipo**: specifica la classe di valori che la variabile può assumere e gli operatori applicabili su tale tipo di dato.
- Vedremo adesso principalmente le seguenti caratteristiche:
  - **Visibilità** (o **scope**): porzione di programma in cui la variabile è nota e può quindi essere utilizzata.
  - **Tempo di vita**: intervallo di tempo in cui resta valida l'associazione tra nome della variabile e indirizzo fisico di memoria.
- Approfondiremo anche i seguenti argomenti:
  - **Specificatori di classe di memorizzazione**: parole chiave che permettono di modificare la classe di memorizzazione di una variabile e/o visibilità di variabili e funzioni.
  - **Qualificatori di tipo**: parole chiave che specificano ulteriori proprietà del tipo di dato.
  - **Organizzazione di un programma**: alcune indicazioni per organizzare un programma costituito da numerose funzioni.

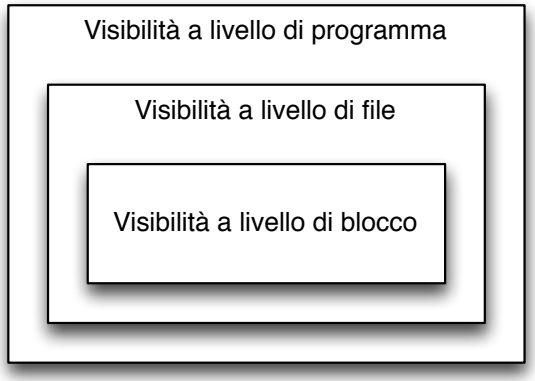
Visibilità di una variabile

- Ogni dichiarazione di variabile nel linguaggio C ha un campo di **visibilità** (o **scope**).
- Una variabile può essere referenziata solo nella regione di programma in cui è visibile.
- All'interno dello stesso scope non ci possono essere due variabili con lo stesso nome.
- In scope distinti, due variabili con lo stesso nome non sono correlate.
- Abbiamo la seguente gerarchia di campi di visibilità



Visibilità di una variabile

- Ogni dichiarazione di variabile nel linguaggio C ha un campo di **visibilità** (o **scope**).
- Una variabile può essere referenziata solo nella regione di programma in cui è visibile.
- All'interno dello stesso scope non ci possono essere due variabili con lo stesso nome.
- In scope distinti, due variabili con lo stesso nome non sono correlate.
- Abbiamo la seguente gerarchia di campi di visibilità



- Anche se qui ci concentriamo sulle variabili, i campi di visibilità si applicano più in generale a tutti gli identificatori in un programma C.
- Il campo di visibilità dipende principalmente dalla posizione della dichiarazione.
- Gli **specificatori di classe di memorizzazione** `extern` e `static` ci permettono di *alterare* la visibilità di variabili e funzioni.

Visibilità a livello di programma: variabili globali

- Le variabili dichiarate al di fuori di ogni funzione sono dette **variabili globali**.
- Sono visibili, e possono essere usate, da tutte le funzioni all'interno dello stesso file.
- Sono visibili alle funzioni definite in altri file del programma (vedremo come).

Visibilità a livello di programma: variabili globali

- Le variabili dichiarate al di fuori di ogni funzione sono dette **variabili globali**.
- Sono visibili, e possono essere usate, da tutte le funzioni all'interno dello stesso file.
- Sono visibili alle funzioni definite in altri file del programma (vedremo come).

```
1  // Esempio di contatore con funzioni e variabile globale
2  #include<stdio.h>
3
4  int count = 0;   // Variabile globale
5  //Setta il valore della variabile globale
6  void set(int x)     { count=x; }
7  // Testa il valore della variabile globale
8  int  lesseq(int x) { return count<=x; }
9  // Incrementa il valore della variabile globale
10 void incr(void)     { ++count; }
11
12 int main() {
13   // Stampa gli interi da 1 a 1000
14   for(set(1); lesseq(1000); incr())
15     printf("%d\n",count);
16   return 0;
17 }
```

- Possiamo utilizzare le variabili globali come *memoria condivisa* tra funzioni.
- Una variabile che possa essere modificata da tutte le funzioni (anche in file esterni) potrebbe rendere il programma difficile da leggere e revisionare.
- Tranne che per casi specifici, l'uso di variabili globali è da evitare, in quanto è generalmente visto come indice di cattivo stile di programmazione.

Visibilità a livello di blocco: variabili locali

- Le variabili dichiarate in un blocco sono dette **variabili locali**, o **automatiche**.
- Sono visibili, e possono essere usate, solo nel blocco in cui sono state dichiarate.
- Una variabile locale **maschera** le variabili con lo stesso nome esterne al blocco.

Visibilità a livello di blocco: variabili locali

- Le variabili dichiarate in un blocco sono dette **variabili locali**, o **automatiche**.
- Sono visibili, e possono essere usate, solo nel blocco in cui sono state dichiarate.
- Una variabile locale **maschera** le variabili con lo stesso nome esterne al blocco.

```c
// Esempio errato di contatore con funzioni e variabili locali
#include<stdio.h>

int count = 0; // Variabile globale

// Non produce effetti all'esterno
void set(int x)    { int count=0; count=x; }
// Ritorna 0 se x<0, 1 altrimenti
int  lesseq(int x) { int count=0; return count<=x; }
// Non produce effetti all'esterno
void incr(void)    { int count=0; ++count; }

int main() {
  // Loop infinito: condizione sempre vera
  for(set(1); lesseq(1000); incr()) {
    int count = 0; // Variabile locale al blocco for
    printf("%d\n",count);
  }
  return 0;
}
```

Tutte le dichiarazioni `int count = 0;` nell'esempio mascherano la variabile globale `count`: ogni riferimento a `count` è un riferimento alla variabile locale al blocco.

Visibilità delle variabili: esempi con errori

- Due variabili nello stesso scope non possono avere lo stesso nome.

```
1 int pos(int x) {
2   int y;
3   int y = 0; // Errore
4   if(x<0) return y;
5   else    return x;
6 }
```

```
1 int pos(int x) {
2   int y = 0;
3   int x = 1; // Errore
4   if(x<0) return y;
5   else    return x;
6 }
```

Visibilità delle variabili: esempi con errori

- Due variabili nello stesso scope non possono avere lo stesso nome.

```c
int pos(int x) {
  int y;
  int y = 0; // Errore
  if(x<0) return y;
  else    return x;
}
```

```c
int pos(int x) {
  int y = 0;
  int x = 1; // Errore
  if(x<0) return y;
  else    return x;
}
```

- Non è possibile referenziare una variabile non visibile nello scope.

```c
int pos(int x) {
  if(x<0) {
    int y = 0;
  } else {
    int y = 1;
  }
  return y; // Errore
}
```

```c
int pos(int x) {
  { // Inizio blocco locale
    int y; // Locale al blocco
    if(x<0) y = 0;
    else    y = 1;
  } // Fine blocco locale
  return y; // Errore
}
```

Lo specificatore di classe di memorizzazione `extern`

- La parola chiave `extern` permette di rende visibile all'interno di un file, funzione o blocco una variabile globale definita in qualche altro punto del programma.

```
extern int x;
```

- La dichiarazione `extern` non causa *allocazione di memoria* per la variabile `x`.
- Indica che ogni riferimento alla variabile `x` nello scope locale è un riferimento ad una variabile globale definita in qualche altro punto del programma.
- Il **linker** (vedi compilazione) si occuperà di *risolvere* il riferimento alla variabile esterna.

Lo specificatore di classe di memorizzazione `extern`

- La parola chiave `extern` permette di rende visibile all'interno di un file, funzione o blocco una variabile globale definita in qualche altro punto del programma.

```
extern int x;
```

- La dichiarazione `extern` non causa *allocazione di memoria* per la variabile `x`.
- Indica che ogni riferimento alla variabile `x` nello scope locale è un riferimento ad una variabile globale definita in qualche altro punto del programma.
- Il **linker** (vedi compilazione) si occuperà di *risolvere* il riferimento alla variabile esterna.
- L'inizializzazione di variabili `extern` può essere interpretata in due modi distinti in fase di compilazione:

```
extern int x = 0;
```

1. la variabile `x` è una variabile globale, definita in qualche altro punto del programma: abbiamo un errore di sintassi;
2. la variabile `x` non è definita in qualche altro punto del programma, quindi l'istruzione sopra è equivalente a

```
int x = 0;
```

Lo specificatore di classe di memorizzazione extern: esempio 1/2

```c
/* file main.c */
#include<stdio.h>
#include "lib.h"

// Variabile globale esterna
extern int count;

int main() {
  for(set(1); lesseq(1000); incr())
    printf("%d\n",count); // Visibile
  return 0;
}
```

```c
/* file lib.h */

//Setta il valore della variabile
void set(int x);

//Testa il valore della variabile
int lesseq(int x);

//Incrementa il valore della variabile
void incr(void);
```

```c
/* File lib.c */

// Variabile globale
int count = 0;

void set(int x) {
  count=x;
}

int lesseq(int x) {
  return count<=x;
}

void incr(void) {
  ++count;
}
```

Il main può accedere alla variabile globale `count`, dichiarata in `lib.c`.

Lo specificatore di classe di memorizzazione extern: esempio 2/2

- Anche se completamente superfluo, è sintatticamente valido referenziare come `extern` una variabile globale nello stesso stesso file.

```
1  #include<stdio.h>
2
3  int count = 0; // Variabile globale
4
5  void set(int x)     {extern int count; count=x; }
6  int  lesseq(int x) {extern int count; return count<=x; }
7  void incr(void)     {extern int count; ++count; }
8
9  int main() {
10   for(set(1); lesseq(1000); incr()) {
11     extern int count;
12     printf("%d\n",count);
13   }
14   return 0;
15 }
```

Tutte le dichiarazioni `extern` nell'esempio fanno riferimento alla variabile globale `count`.

Dichiarazione vs definizione: precisazioni

- Citiamo direttamente lo standard ISO C89 relativamente alla differenza tra **dichiarazione** e **definizione** nel linguaggio C:
  A declaration specifies the interpretation and attributes of a set of identifiers. A declaration that also causes storage to be reserved for an object or function named by an identifier is a definition.

Dichiarazione vs definizione: precisazioni

- Citiamo direttamente lo standard ISO C89 relativamente alla differenza tra **dichiarazione** e **definizione** nel linguaggio C:
  A declaration specifies the interpretation and attributes of a set of identifiers. A declaration that also causes storage to be reserved for an object or function named by an identifier is a definition.
- Per le funzioni abbiamo utilizzato una terminologia molto precisa:
  - **Dichiarazione di funzione**: dichiariamo il prototipo della funzione (tipo di ritorno, numero e tipo di argomenti). Una dichiarazione di funzione non richiede memoria da riservare in fase di esecuzione del programma.
  - **Definizione di funzione**: definiamo il codice della funzione. Una definizione richiede spazio di memoria per contenere le istruzioni eseguibili della funzione.

Dichiarazione vs definizione: precisazioni

- Citiamo direttamente lo standard ISO C89 relativamente alla differenza tra **dichiarazione** e **definizione** nel linguaggio C:
  A declaration specifies the interpretation and attributes of a set of identifiers. A declaration that also causes storage to be reserved for an object or function named by an identifier is a definition.
- Per le funzioni abbiamo utilizzato una terminologia molto precisa:
  - **Dichiarazione di funzione**: dichiariamo il prototipo della funzione (tipo di ritorno, numero e tipo di argomenti). Una dichiarazione di funzione non richiede memoria da riservare in fase di esecuzione del programma.
  - **Definizione di funzione**: definiamo il codice della funzione. Una definizione richiede spazio di memoria per contenere le istruzioni eseguibili della funzione.
- Per le variabili siamo stati (inevitabilmente) più vaghi. Chiariamo adesso come si applica questa terminologia alle variabili.
  - **Dichiarazione di variabile**: dichiariamo l'esistenza della variabile, senza definirla. Non viene riservata memoria per la variabile.

```
extern int x; // Dichiarazione
```

  - **Definizione di variabile**: dichiariamo l'esistenza della variabile e richiediamo spazio di memoria da riservare alla variabile.

```
extern int x = 0; // Dichiarazione e definizione
int y;            // Dichiarazione e definizione
```

- Sono ammesse dichiarazioni multiple della stessa variabile ma una sola definizione nello stesso scope.

Lo specificatore di classe di memorizzazione `static`

- La parola chiave `static` permette di modificare la *visibilità* di dichiarazioni di variabili globali e funzioni.
- Le variabili globali e funzioni `static` sono visibili unicamente a livello di file.
- Le variabili `static`, se non esplicitamente inizializzate, sono implicitamente inizializzate a zero (più precisamente, tutti i bit della locazione di memoria corrispondente sono inizializzati a zero).

Lo specificatore di classe di memorizzazione `static`

- La parola chiave `static` permette di modificare la *visibilità* di dichiarazioni di variabili globali e funzioni.
- Le variabili globali e funzioni `static` sono visibili unicamente a livello di file.
- Le variabili `static`, se non esplicitamente inizializzate, sono implicitamente inizializzate a zero (più precisamente, tutti i bit della locazione di memoria corrispondente sono inizializzati a zero).
- Lo specificatore `static` permette inoltre di modificare le *modalità di memorizzazione* di una variabile locale. Vedremo la sua semantica in questo contesto quando parleremo del tempo di vita delle variabili.
- Ci concentriamo adesso sulla semantica della parla chiave `static` in presenza di:
  - dichiarazioni di variabili globali;
  - dichiarazioni/definizioni di funzioni.

Lo specificatore di classe di memorizzazione `static`: variabili globali

- Se applichiamo lo specificatore `static` ad una variabile globale, tale variabile diventa nota unicamente alle funzioni presenti nello stesso file.

```c
/* file main.c */
#include<stdio.h>
#include "lib.h"

int main() {
  for(set(1); lesseq(1000); incr())
    printf("%d\n",val());
  return 0;
}
```

```c
/* file lib.h */
//Setta il valore della variabile
void set(int x);

//Testa il valore della variabile
int lesseq(int x);

//Incrementa il valore della variabile
void incr(void);

// Ritorna il valore della variabile
int val();
```

```c
/* File lib.c */

// Variabile globale
// a livello di file
static int count=0;

void set(int x) {
  count=x;
}

int lesseq(int x) {
  return count<=x;
}

void incr(void) {
  ++count;
}

int val() {
  return count;
}
```

Il `main()` non può referenziare direttamente la variabile `count` (nemmeno con `extern`).

Lo specificatore di classe di memorizzazione static: funzioni

- Una funzione static è nota unicamente alle funzioni presenti nello stesso file.

```c
/* file main.c */
#include<stdio.h>
#include "lib.h"

int main() {
  for(set(1); lesseq(1000); incr())
    printf("%d\n",val());
  return 0;
}
```

```c
/* file lib.h */
// Setta il valore della variabile
void set(int x);

// Testa il valore della variabile
int lesseq(int x);

// Incrementa il valore della variabile
// e ritorna 1. Ritorna 0 se non pu
// essere incrementata.
int incr(void);

// Ritorna il valore della variabile
int val();
```

```c
/* File lib.c */
#include <limits.h>
static int count=0;

static int maxval(void) {
 return count==INT_MAX;
}

void set(int x) {
  count=x;
}

int lesseq(int x) {
  return count<=x;
}

int incr(void) {
  if(maxval()) return 1;
  ++count;
  return 0;
}

int val() {
  return count;
}
```

Il `main()` non può chiamare la funzione `maxval()`. Darebbe errore in fase di compilazione.

Visibilità di variabili e funzioni: riassunto

- Variabili:
  - Una variabile globale è visibile a livello di programma.
  - Una variabile globale `static` è visibile solo a livello di file.
  - Una variabile locale è visibile solo nel blocco in cui è stata dichiarata.

Visibilità di variabili e funzioni: riassunto

- Variabili:
  - Una variabile globale è visibile a livello di programma.
  - Una variabile globale `static` è visibile solo a livello di file.
  - Una variabile locale è visibile solo nel blocco in cui è stata dichiarata.
- Funzioni
  - Una funzione è visibile alle funzioni che seguono la sua dichiarazione/definizione.
  - Una funzione nello stesso programma può essere invocata anche se non visibile, a patto che il suo tipo di ritorno sia `int` (sintassi K&R).
  - Una funzione `static` è visibile solo a livello di file.

Visibilità di variabili e funzioni: riassunto

- Variabili:
  - Una variabile globale è visibile a livello di programma.
  - Una variabile globale `static` è visibile solo a livello di file.
  - Una variabile locale è visibile solo nel blocco in cui è stata dichiarata.
- Funzioni
  - Una funzione è visibile alle funzioni che seguono la sua dichiarazione/definizione.
  - Una funzione nello stesso programma può essere invocata anche se non visibile, a patto che il suo tipo di ritorno sia `int` (sintassi K&R).
  - Una funzione `static` è visibile solo a livello di file.

```
1  #include<stdio.h>
2  int x = 0;              // Variabile visibile a livello di programma
3  static int y = 0;       // Variabile visibile a livello di file
4
5  static int f(void) { // Funzione visibile a livello di file
6    return 10;
7  }
8
9  int g(void) {           // Funzione invocabile a livello di programma
10   int i;                // Variabile visibile a livello di blocco (funzione)
11   for(i=x; i<=f(); i++) {
12     int j = y-i;        // Variabile visibile a livello di blocco
13     printf("%d\n",j);
14   }
15   return i;
16 }
```

Tempo di vita di una variabile

- Una variabile, oltre ad avere un campo di visibilità, ha un **tempo di vita**.
- Il tempo di vita di una variabile indica l'intervallo di tempo in cui esiste un'area di memoria associata alla variabile.

Tempo di vita di una variabile

- Una variabile, oltre ad avere un campo di visibilità, ha un **tempo di vita**.
- Il tempo di vita di una variabile indica l'intervallo di tempo in cui esiste un'area di memoria associata alla variabile.
- Abbiamo due classi di **storage duration** (tempo di vita):
  1. **static**. Il tempo di vita si estende per tutta la durata del programma.
  2. **automatic**. Il tempo di vita è legato all'ambito di visibilità locale della variabile.

Tempo di vita di una variabile

- Una variabile, oltre ad avere un campo di visibilità, ha un **tempo di vita**.
- Il tempo di vita di una variabile indica l'intervallo di tempo in cui esiste un'area di memoria associata alla variabile.
- Abbiamo due classi di **storage duration** (tempo di vita):
  1. **static**. Il tempo di vita si estende per tutta la durata del programma.
  2. **automatic**. Il tempo di vita è legato all'ambito di visibilità locale della variabile.
- Dal punto di vista delle variabili, abbiamo:
  - **Variabile locale** (tempo di vita **automatic** o **static**).
    - Viene *creata* al momento della dichiarazione.
    - Viene *distrutta* non appena termina il blocco in cui è stata dichiarata.
    - E' possibile possibile modificare il tempo di vita della variabile locale da **automatic** a **static** facendo uso dello specificatore `static`.
  - **Variabile globale** (tempo di vita **static**).
    - Viene *creata* prima di iniziare l'esecuzione del programma.
    - Il suo tempo di vita si estende per tutta la durata del programma.

Tempo di vita di una variabile locale `static`

- La parola chiave `static` se utilizzata con variabili locali, istruisce il compilatore di mantenere in vita la variabile per l'intera durata del programma.
- Una variabile locale `static` preserva il proprio contenuto di memoria anche quando il blocco in cui è dichiarata termina.

Tempo di vita di una variabile locale `static`

- La parola chiave `static` se utilizzata con variabili locali, istruisce il compilatore di mantenere in vita la variabile per l'intera durata del programma.
- Una variabile locale `static` preserva il proprio contenuto di memoria anche quando il blocco in cui è dichiarata termina.
- E' molto utile per preservare il valore di una variabile locale tra chiamate successive a funzione.

```
1  #include <stdio.h>
2
3  // Semplice generatore lineare congruenziale
4  // di numeri casuali tra 0 e 15
5  int myrand(void) {
6    static int seed = 1;
7    return seed = (5*seed+3)%16;
8  }
9
10 int main() {
11   int i;
12
13   // Stampa100 numeri random tra 0 e 15
14   for(i=0; i<100; i++)
15     printf("%d\n",myrand());
16   return 0;
17 }
```

La variabile `seed` nella funzione `myrand()` viene inizializzata solo alla prima chiamata.

Specificatori di classe di memorizzazione

- Le quattro parole chiave `extern`, `static`, `register`, `auto` sono **specificatori di classe di memorizzazione**.
- Modificano il tempo di vita e la visibiltà di variabili e funzioni in un programma C.
- Abbiamo già discusso l'uso degli specificatori `extern` e `static`. Riassumiamo la loro semantica e descriviamo i due ulteriori specificatori `register` e `auto`.

Specificatori di classe di memorizzazione

- Le quattro parole chiave `extern`, `static`, `register`, `auto` sono **specificatori di classe di memorizzazione**.
- Modificano il tempo di vita e la visibiltà di variabili e funzioni in un programma C.
- Abbiamo già discusso l'uso degli specificatori `extern` e `static`. Riassumiamo la loro semantica e descriviamo i due ulteriori specificatori `register` e `auto`.
  - `extern`. Permette di referenziare una variabile globale la cui dichiarazione potrebbe risiedere in un file esterno.

Specificatori di classe di memorizzazione

- Le quattro parole chiave `extern`, `static`, `register`, `auto` sono **specificatori di classe di memorizzazione**.
- Modificano il tempo di vita e la visibiltà di variabili e funzioni in un programma C.
- Abbiamo già discusso l'uso degli specificatori `extern` e `static`. Riassumiamo la loro semantica e descriviamo i due ulteriori specificatori `register` e `auto`.
  - `extern`. Permette di referenziare una variabile globale la cui dichiarazione potrebbe risiedere in un file esterno.
  - `static`. Utilizzato con una variabile globale o funzione, limita lo scope della variabile/funzione a livello di file. Utilizzato con una variabile locale, estende il tempo di vita della variabile all'intera durata del programma.

Specificatori di classe di memorizzazione

- Le quattro parole chiave `extern`, `static`, `register`, `auto` sono **specificatori di classe di memorizzazione**.
- Modificano il tempo di vita e la visibiltà di variabili e funzioni in un programma C.
- Abbiamo già discusso l'uso degli specificatori `extern` e `static`. Riassumiamo la loro semantica e descriviamo i due ulteriori specificatori `register` e `auto`.
  - `extern`. Permette di referenziare una variabile globale la cui dichiarazione potrebbe risiedere in un file esterno.
  - `static`. Utilizzato con una variabile globale o funzione, limita lo scope della variabile/funzione a livello di file. Utilizzato con una variabile locale, estende il tempo di vita della variabile all'intera durata del programma.
  - `register`. *Suggerisce* al compilatore di rendere l'accesso alla variabile il più rapido possibile (ad esempio, tramite salvataggio nei registri del processore, invece che in RAM). Il compilatore può decidere di ignorare il suggerimento.

Specificatori di classe di memorizzazione

- Le quattro parole chiave `extern`, `static`, `register`, `auto` sono **specificatori di classe di memorizzazione**.
- Modificano il tempo di vita e la visibiltà di variabili e funzioni in un programma C.
- Abbiamo già discusso l'uso degli specificatori `extern` e `static`. Riassumiamo la loro semantica e descriviamo i due ulteriori specificatori `register` e `auto`.
  - `extern`. Permette di referenziare una variabile globale la cui dichiarazione potrebbe risiedere in un file esterno.
  - `static`. Utilizzato con una variabile globale o funzione, limita lo scope della variabile/funzione a livello di file. Utilizzato con una variabile locale, estende il tempo di vita della variabile all'intera durata del programma.
  - `register`. *Suggerisce* al compilatore di rendere l'accesso alla variabile il più rapido possibile (ad esempio, tramite salvataggio nei registri del processore, invece che in RAM). Il compilatore può decidere di ignorare il suggerimento.
  - `auto`. E' il tipo di default delle variabili locali (chiamate anche variabili automatiche). Specificatore superfluo, dato che le uniche dichiarazioni in cui è sintatticamente corretto utilizzarlo sono di default `auto`.

Specificatori di classe di memorizzazione

- Le quattro parole chiave `extern`, `static`, `register`, `auto` sono **specificatori di classe di memorizzazione**.
- Modificano il tempo di vita e la visibiltà di variabili e funzioni in un programma C.
- Abbiamo già discusso l'uso degli specificatori `extern` e `static`. Riassumiamo la loro semantica e descriviamo i due ulteriori specificatori `register` e `auto`.
  - `extern`. Permette di referenziare una variabile globale la cui dichiarazione potrebbe risiedere in un file esterno.
  - `static`. Utilizzato con una variabile globale o funzione, limita lo scope della variabile/funzione a livello di file. Utilizzato con una variabile locale, estende il tempo di vita della variabile all'intera durata del programma.
  - `register`. *Suggerisce* al compilatore di rendere l'accesso alla variabile il più rapido possibile (ad esempio, tramite salvataggio nei registri del processore, invece che in RAM). Il compilatore può decidere di ignorare il suggerimento.
  - `auto`. E' il tipo di default delle variabili locali (chiamate anche variabili automatiche). Specificatore superfluo, dato che le uniche dichiarazioni in cui è sintatticamente corretto utilizzarlo sono di default `auto`.
- In una dichiarazione è possibile utilizzare al massimo uno specificatore di classe di memorizzazione.
  - Non possiamo dichiarare una variabile con classe di memorizzazione `extern` `static`.

Qualificatori di tipo

- I **qualificatori di tipo**, `const` e `volatile`, forniscono proprietà aggiuntive ai tipi di dato visti finora:
  - `const`. Indica un tipo di dato il cui contenuto non può essere modificato del programma, cioè un tipo di dato *read-only* (a sola lettura);
  - `volatile`. Indica un tipo di dato che può essere modificato da procedure esterne al programma stesso, come sistema operativo e hardware (ad esempio, orologio di sistema, interrupt, ecc.).

```
1 // Pi constant
2 const float pi = 3.14;
3 // Hardware signal
4 volatile int hw_signal;
```

Qualificatori di tipo

- I **qualificatori di tipo**, `const` e `volatile`, forniscono proprietà aggiuntive ai tipi di dato visti finora:
  - `const`. Indica un tipo di dato il cui contenuto non può essere modificato del programma, cioè un tipo di dato *read-only* (a sola lettura);
  - `volatile`. Indica un tipo di dato che può essere modificato da procedure esterne al programma stesso, come sistema operativo e hardware (ad esempio, orologio di sistema, interrupt, ecc.).

```
1 // Pi constant
2 const float pi = 3.14;
3 // Hardware signal
4 volatile int hw_signal;
```

- I qualificatori di tipo possono essere applicati a tutte le combinazioni ammissibili di specificatori di tipo, modificatori di tipo e specificatori di classe di memorizzazione.

Qualificatori di tipo

- I **qualificatori di tipo**, `const` e `volatile`, forniscono proprietà aggiuntive ai tipi di dato visti finora:
  - `const`. Indica un tipo di dato il cui contenuto non può essere modificato del programma, cioè un tipo di dato *read-only* (a sola lettura);
  - `volatile`. Indica un tipo di dato che può essere modificato da procedure esterne al programma stesso, come sistema operativo e hardware (ad esempio, orologio di sistema, interrupt, ecc.).

```
1 // Pi constant
2 const float pi = 3.14;
3 // Hardware signal
4 volatile int hw_signal;
```

- I qualificatori di tipo possono essere applicati a tutte le combinazioni ammissibili di specificatori di tipo, modificatori di tipo e specificatori di classe di memorizzazione.
- La seguente dichiarazione di variabile, per quanto controintuitiva, è sintatticamente e semanticamente valida in C:

```
1 int const volatile flag;
```

La variabile *flag* contiene un tipo di dato *intero* a *sola lettura* all'interno del programma ma che può essere modificato dall'esterno. Un esempio potrebbe essere un canale di comunicazione a sola lettura per dialogare con qualche driver del sistema operativo.

Organizzazione dei file sorgenti 1/2

- Come organizziamo un programma che consiste di numerose funzioni?

Organizzazione dei file sorgenti 1/2

- Come organizziamo un programma che consiste di numerose funzioni?
- **Librerie**.
  - Raggruppiamo in file differenti (librerie) le funzioni che hanno un *tema* comune.
  - Ad esempio: libreria di funzioni matematiche, funzioni di I/O, ecc.
  - *Nascondiamo* con la parola chiave `static` le funzioni/utility interne alla libreria che non vogliamo mostrare all'esterno.
  - *Nascondiamo* con la parola chiave `static` le variabili globali.
  - I punti *critici* delle funzioni devono essere commentati.

Organizzazione dei file sorgenti 1/2

- Come organizziamo un programma che consiste di numerose funzioni?
- **Librerie**.
  - Raggruppiamo in file differenti (librerie) le funzioni che hanno un *tema* comune.
  - Ad esempio: libreria di funzioni matematiche, funzioni di I/O, ecc.
  - *Nascondiamo* con la parola chiave `static` le funzioni/utility interne alla libreria che non vogliamo mostrare all'esterno.
  - *Nascondiamo* con la parola chiave `static` le variabili globali.
  - I punti *critici* delle funzioni devono essere commentati.
- **Header**
  - Ogni file di *libreria* (`lib.c`) deve essere associato ad un file *header* (`lib.h`).
  - L'header deve contenere i prototipi delle funzioni nella libreria corrispondente.
  - Deve contenere solo **dichiarazioni**, non definizioni o codice eseguibile.
  - I prototipi e le eventuali dichiarazioni devono essere commentati: deve essere possibile capire come utilizzare una libreria consultando il suo header.
  - E' sempre utile *proteggere* il contenuto dell'header da inclusioni multiple, facendo uso della direttiva del preprocessore `#ifndef`.

Organizzazione dei file sorgenti 1/2

- Come organizziamo un programma che consiste di numerose funzioni?
- **Librerie**.
  - Raggruppiamo in file differenti (librerie) le funzioni che hanno un *tema* comune.
  - Ad esempio: libreria di funzioni matematiche, funzioni di I/O, ecc.
  - *Nascondiamo* con la parola chiave `static` le funzioni/utility interne alla libreria che non vogliamo mostrare all'esterno.
  - *Nascondiamo* con la parola chiave `static` le variabili globali.
  - I punti *critici* delle funzioni devono essere commentati.
- **Header**
  - Ogni file di *libreria* (`lib.c`) deve essere associato ad un file *header* (`lib.h`).
  - L'header deve contenere i prototipi delle funzioni nella libreria corrispondente.
  - Deve contenere solo **dichiarazioni**, non definizioni o codice eseguibile.
  - I prototipi e le eventuali dichiarazioni devono essere commentati: deve essere possibile capire come utilizzare una libreria consultando il suo header.
  - E' sempre utile *proteggere* il contenuto dell'header da inclusioni multiple, facendo uso della direttiva del preprocessore `#ifndef`.
- **File principale**
  - Il file principale contiene la funzione `main()`.
  - Dovrebbe gestire solo i parametri di input e l'interfaccia del programma.
  - Deve includere i file header delle librerie di cui fa uso.

Organizzazione dei file sorgenti 2/2

- Come organizziamo il codice in un singolo file del programma?

Organizzazione dei file sorgenti 2/2

- Come organizziamo il codice in un singolo file del programma?
- **Commenti iniziali**
  - Se necessario, il file deve iniziare con un commento che ne descriva il contenuto e che contenga altre informazioni come: autore, versione, copyright, ecc.

Organizzazione dei file sorgenti 2/2

- Come organizziamo il codice in un singolo file del programma?
- **Commenti iniziali**
  - Se necessario, il file deve iniziare con un commento che ne descriva il contenuto e che contenga altre informazioni come: autore, versione, copyright, ecc.
- **Direttive al preprocessore**.
  - Le direttive devono essere specificate prima di dichiarazioni e definizioni.
  - Deve contenere direttive di inclusione solo per i file strettamente necessari.

Organizzazione dei file sorgenti 2/2

- Come organizziamo il codice in un singolo file del programma?
- **Commenti iniziali**
  - Se necessario, il file deve iniziare con un commento che ne descriva il contenuto e che contenga altre informazioni come: autore, versione, copyright, ecc.
- **Direttive al preprocessore**.
  - Le direttive devono essere specificate prima di dichiarazioni e definizioni.
  - Deve contenere direttive di inclusione solo per i file strettamente necessari.
- **Dichiarazioni**
  - Le dichiarazioni di variabili esterne, strutture, tipi definiti dal programmatore, ecc., devono precedere le definizioni.

Organizzazione dei file sorgenti 2/2

- Come organizziamo il codice in un singolo file del programma?
- **Commenti iniziali**
  - Se necessario, il file deve iniziare con un commento che ne descriva il contenuto e che contenga altre informazioni come: autore, versione, copyright, ecc.
- **Direttive al preprocessore**.
  - Le direttive devono essere specificate prima di dichiarazioni e definizioni.
  - Deve contenere direttive di inclusione solo per i file strettamente necessari.
- **Dichiarazioni**
  - Le dichiarazioni di variabili esterne, strutture, tipi definiti dal programmatore, ecc., devono precedere le definizioni.
- **Prototipi**
  - I prototipi delle funzioni devono precedere la definizione delle stesse.
  - Non è necessario dichiarare i prototipi delle funzioni che sono visibili all'esterno, se sono stati precedentemente inclusi con la direttiva `#include`.

Organizzazione dei file sorgenti 2/2

- Come organizziamo il codice in un singolo file del programma?
- **Commenti iniziali**
  - Se necessario, il file deve iniziare con un commento che ne descriva il contenuto e che contenga altre informazioni come: autore, versione, copyright, ecc.
- **Direttive al preprocessore**.
  - Le direttive devono essere specificate prima di dichiarazioni e definizioni.
  - Deve contenere direttive di inclusione solo per i file strettamente necessari.
- **Dichiarazioni**
  - Le dichiarazioni di variabili esterne, strutture, tipi definiti dal programmatore, ecc., devono precedere le definizioni.
- **Prototipi**
  - I prototipi delle funzioni devono precedere la definizione delle stesse.
  - Non è necessario dichiarare i prototipi delle funzioni che sono visibili all'esterno, se sono stati precedentemente inclusi con la direttiva `#include`.
- **Definizioni di variabili**
  - Le definizioni di variabili globali devono precedere le definizioni di funzioni.
  - Limitare l'uso di variabili globali o, in alternativa, limitarne la visibilità al file.

Organizzazione dei file sorgenti 2/2

- Come organizziamo il codice in un singolo file del programma?
- **Commenti iniziali**
  - Se necessario, il file deve iniziare con un commento che ne descriva il contenuto e che contenga altre informazioni come: autore, versione, copyright, ecc.
- **Direttive al preprocessore**.
  - Le direttive devono essere specificate prima di dichiarazioni e definizioni.
  - Deve contenere direttive di inclusione solo per i file strettamente necessari.
- **Dichiarazioni**
  - Le dichiarazioni di variabili esterne, strutture, tipi definiti dal programmatore, ecc., devono precedere le definizioni.
- **Prototipi**
  - I prototipi delle funzioni devono precedere la definizione delle stesse.
  - Non è necessario dichiarare i prototipi delle funzioni che sono visibili all'esterno, se sono stati precedentemente inclusi con la direttiva `#include`.
- **Definizioni di variabili**
  - Le definizioni di variabili globali devono precedere le definizioni di funzioni.
  - Limitare l'uso di variabili globali o, in alternativa, limitarne la visibilità al file.
- **Definizioni di funzioni**
  - Le definizioni di funzioni devono occupare la parte finale del file.
  - Devono essere nascoste all'esterno le funzioni di utility (*funzioni private*) che hanno il solo scopo di semplificare la definizione delle funzioni che vogliamo mostrare all'esterno (*funzioni pubbliche*).
  - Per convenzione, nel file principale, il `main()` viene definito come prima funzione.

Organizzazione dei file sorgenti: esempio

File principale

Direttive al preprocessore

```
#include <stdio.h>
#include "lib.h"
```

Dichiarazioni

Prototipi

Definizioni di variabili

main

```
int main() {
  for(set(1); lesseq(1000); incr())
    printf("%d\n",val());
  return 0;
}
```

Definizioni di funzioni

lib.c

Direttive al preprocessore

```
#include <limits.h>
#include "lib.h"
```

Dichiarazioni

Prototipi

```
static int maxval(void);
```

Definizioni di variabili

```
static int count = 0;
```

Definizioni di funzioni "pubbliche"

```
void set(int x) { count=x; }

int val() { return count; }

int lesseq(int x) {
  return count <=x;
}

int incr(void) {
  // Verifica potenziali overflow
  if(maxval()) return 0;
  ++count;
  return 1;
}
```

Definizioni di funzioni "private"

```
static int maxval(void) {
  return count==INT_MAX;
}
```

lib.h

Direttive al preprocessore

```
#ifndef LIB_H
#define LIB_H
```

Dichiarazioni

Prototipi di funzioni "pubbliche"

```
// Setta il valore della variabile
void set(int x);

// Testa il valore della variabile
int lesseq(int x);

// Incrementa il valore della
// variabile e ritorna 1. Ritorna 0
//se non puo' essere incrementata.
int incr(void);

// Ritorna il valore della variabile
int val();
```

```
#endif
```