

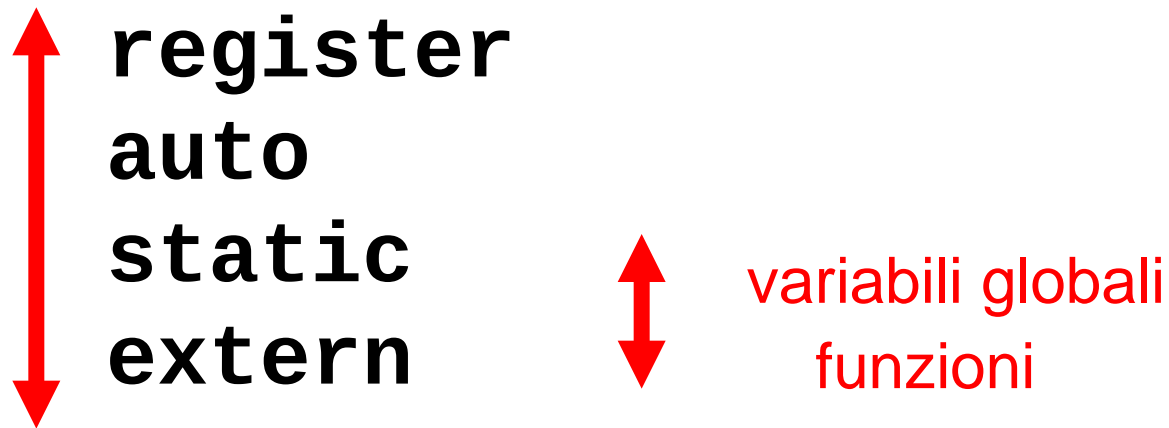
Classi di memorizzazione

Classi di memorizzazione

- Definiscono le regole di visibilità delle variabili e delle funzioni quando il programma è diviso su più file
- Stabiliscono dove (in quale parte dello spazio di indirizzamento) vengono allocate le varie variabili
- Stabiliscono cosa finisce nella tabella dei simboli del modulo oggetto
 - simboli (esterni, esportati)

Classi di memorizzazione (2)

- Tutti i modificatori possono essere applicati alle variabili locali
- Solo **static** ed **extern** a globali e funzioni



variabili
locali

Variabili locali

- La classe di memorizzazione di default per le variabili locali è **auto** (automatica):

```
int x;
```

```
auto int x;
```

- significano entrambe la stessa cosa : la variabile viene allocata sullo stack all'ingresso del blocco in cui è dichiarata e rimossa all'uscita
- non c'è alcun legame fra il valore ad una certa attivazione ed il valore alla attivazione successiva
- la visibilità di **x** è solo all'interno del blocco

Variabili locali (2)

- La classe di memorizzazione **register** è obsoleta

```
int x;
```

```
register int x;
```

- significa che la variabile **x** è usata pesantemente e consiglia al compilatore di metterla in un registro generale
- viene ignorata, i compilatori di oggi sono dotati di sofisticati algoritmi di allocazione dei registri
- la visibilità di **x** è solo all'interno del blocco

Variabili locali (3)

- La classe di memorizzazione **static** è utilizzata quando il valore della variabile deve essere mantenuto da una invocazione all'altra del blocco

```
{ static int x;  
...  
}
```

- significa che la variabile **x** mantiene il proprio valore da una attivazione del blocco alla successiva
- la visibilità di **x** è solo all'interno del blocco

Variabili locali (4)

- La classe di memorizzazione **extern**

```
{ extern int x;
```

```
...
```

```
}
```

- significa che la variabile **x** è dichiarata globale in questo o in altri file
- il nome **x** finisce nella *tabella dei simboli come simbolo esterno*
- il compilatore segnala la situazione nell'oggetto ed il linker 'cerca **x** altrove'

Variabili globali

- La classe di memorizzazione **extern** è la condizione di default per le variabili globali

```
int x;
```

```
extern int x;
```

```
int funct (...) { ... }
```

- queste due dichiarazioni di **x** sono equivalenti, ed **x** viene allocata nell'area dati statica
- il nome **x** finisce nella *tabella dei simboli come simbolo esportato*
- la visibilità di **x** comprende tutte le funzioni dichiarate nel file dopo di essa ed **x** può essere utilizzata in un altro file

Variabili globali (2)

- La classe di memorizzazione **static** serve a limitare la portata delle variabili globali

```
static int x;
```

```
int funct (...) { ... }
```

- questa dichiarazione di **x** rende la variabile visibile da tutte le funzioni che la seguono nel file
- MA, il nome **x** NON finisce nella *tabella dei simboli esportati*
- la visibilità di **x** è ristretta al file dove è dichiarata
- può servire per realizzare variabili ‘private’ di una implementazione

Funzioni

- **extern** è la classe di memorizzazione di default delle funzioni

```
int funct (...) { ... }
```

```
extern int funct (...) { ... }
```

- sono equivalenti : **funct** è chiamabile all'interno di tutte le funzioni che la seguono nel file
- il nome **funct** finisce nella *tabella dei simboli come simbolo esportato*
- funzioni di altri file possono usare **funct**, chiamate a funzioni definite in altri file finiscono nella *tabella dei simboli come esterne* e vengono risolte dal linker (es. printf)

Funzioni (2)

- La classe di memorizzazione **static** serve a limitare la portata delle funzioni

```
static int funct (...) { ... }
```

- questa dichiarazione di **funct** la rende visibile *solo* a tutte le funzioni che la seguono nel file
- MA, il nome **funct** NON finisce nella *tabella dei simboli esportati*
- può servire per implementare delle funzioni ‘private’ di una implementazione, non chiamabili dall’esterno del file

Realizzare una libreria in C

- Idea di base :
 - definire l'interfaccia in un file `X.h` (tipi di dato e prototipi delle funzioni utilizzabili)
 - utilizzare lo specificatore `static` per realizzare la protezione nel file `X.c` che realizza l'implementazione
 - mettere a disposizione l'oggetto `X.o` (da solo o all'interno di una libreria) per permettere un linking corretto
- Esempio : il tipo **liste**

Realizzare una libreria in C (2)

- ... Come si usa il tipo liste
 - includere l'header (**liste.h**) nel file che usa le funzioni ed i tipi della libreria
 - compilare linkando anche **liste.o** es:
gcc -o ese liste.o main.c
- Insiemi di file oggetti di uso comune possono essere raccolti in librerie (estensione .a)

Librerie (ar)

- Il comando :

ar r nomebibreria.a file1.o fileN.o

– inserisce/aggiorna i file **file1.o fileN.o** nella libreria **nomebibreria.a**

– es.

ar r libliste.a liste.o listeextra.o

crea una libreria contenente i due oggetti

- **r** : operazione da eseguire

- **nm -s libreria.a**

fornisce informazioni sui simboli definiti nei vari file della libreria (con indicazioni del file dove si trovano)

Librerie (ar) (2)

- Tipicamente le librerie vengono raccolte in una *directory*

- es. `~/lib/`

- e recuperate in fase di linking per generare l'eseguibile finale

- es: `gcc -L~/lib/ -lliste main.c`

Compila `main.c` e lo *collega* con le funzioni nella libreria `libliste.a` in `~/lib/`