VARIABILI LOCALI E GLOBALI (ESTERNE)

- Le variabili ***locali*** sono **definite** (ed il loro uso **dichiarato)** nella funzione (o nel blocco) che le usa; nascono quando la funzione entra in esecuzione e muoiono al termine dell'esecuzione della funzione.
- La **definizione** di una variabile ***globale*** vale dal punto di definizione fino alla fine del file. Lo spazio viene allocato ed inizializzato per default a zero, a meno che sia diversamente inizializzato dal programmatore.
  Se la definizione di una variabile globale precede, nel file, quella di una funzione allora la funzione può tranquillamente usare la variabile. Altrimenti la funzione deve **dichiararne** l'uso, premettendo ***extern*** nella dichiarazione della variabile dentro la funzione. Ci deve essere una sola definizione ma ci possono essere più dichiarazioni *extern* da parte di più funzioni. **Pratica comune è premettere in un file tutte le definizioni di variabili globali e poi omettere tutte le dichiarazioni di *extern* nelle funzioni del file**.
- Se un programma è su più files, e ad esempio una variabile globale è definita in file1 ed usata in file2 e in file3, è necessario dichiarare la variabile *extern* in file2 e file3.
- È bene usare poche variabili globali perchè:

  - restano in vita sempre, anche se non servono più;
  - possono essere cambiate in modo inaspettato da più funzioni;
  - possono distruggere la generalità di certe funzioni, legandole ai nomi delle variabili globali che esse manipolano.

Le variabili hanno un ambito di visibilità (detto scope). L'ambito di visibilità denota la parte del testo sorgente C in cui la sua dichiarazione è attiva.

Le variabili hanno anche una durata, che descrive il lasso temporale di memorizzazione dei valori di una variabile:

Le proprietà di visibilità e di durata individuano la classe di memorizzazione di una variabile.

Regole di Visibilità (SCOPE)
di una variabile (o di un dentificatore)

Definiscono l'ambiente del programma dove la dichiarazione è attiva e dove la variabile può quindi essere referenziata.

- Visibilità a livello di blocco (**Block scope)** Per blocco si intende un qualunque insieme di istruzioni delimitate da {} (corpo di una funzione, istruzioni composte, ecc.). La visibilità inizia dal punto del blocco in cui è dichiarata fino alla fine del blocco, fine indicata da }.
  Esempi: variabili locali, parametri di funzioni (considerati variabili locali di funzioni), variabili con lo stesso nome in blocchi esterni "nascosti" da blocchi più interni, ecc.

- Visibilità a livello di funzione (**Function scope)** Una variabile è accessibile dall'inizio alla fine della funzione in cui è dichiarata. I *labels* sono gli unici identificatori con questa visibilità (start:, case:, etc.). Possono essere referenziati solo dentro il corpo di una funzione.

- Visibilità a livello di file (**File scope)** Quando una variabile è definita fuori da ogni funzione essa è accessibile dal punto in cui è dichiarata fino alla fine del file in cui si trova la dichiarazione. È "visibile" da tutte le funzioni dal punto in cui è dichiarata fino alla fine del file. *Esempi*: variabili globali, definizioni di funzioni e prototipi di funzioni.

- Visibilità a livello di prototipo (**Function prototype scope)** Quando sono nella lista dei parametri di un prototipo di funzione.

CLASSI DI MEMORIA(Storage classes)

Quattro **specificatori** indicano come una variabile viene allocata in memoria. **Extern** e **Static** indicano allocazione statica, **auto** e **register** indicano allocazione temporanea.

- **Storage statico.** Variabile esiste per l'intera esecuzione del programma.

  extern Di default per variabili globali e per funzioni.
  static Il prefisso static, **applicato ad una variabile globale** (o anche ad una funzione), limita la sua conoscenza all'interno del file dove è definita, nascondendola a tutti gli altri files (extern non funziona). Invece, **applicato ad una variabile locale**, la rende permanente, non temporanea. Variabili locali static mantengono il loro valore dopo la fine della funzione.

Es:
```
int conta()
{
    static int counter = 1 ;
    printf("Funzione eseguita %d volte\n",counter++);
}
```

- **Storage automatico.** Variabile creata e distrutta nel suo blocco; ha vita temporanea.

  auto Di default per variabili locali.
  register Usabile solo con variabili locali e parametri. La parola chiave register consente di suggerire al compilatore quali variabili dovrebbero essere memorizzate nei registri. Il livello di supporto offerto dai compilatori è molto variabile: alcuni compilatori memorizzano tutte le variabili register in registri, fino a quando ce ne sono disponibili, altri lo ignorano, altri lo interpretano per determinare se è davvero proficuo memorizzare una data variabile in un registro. Ad una variabile register non è assegnato alcun indirizzo di memoria: anche se il suggerimento register non viene seguito dal compilatore, se si tenta di accedere all'indirizzo della variabile, si ottiene una segnalazione di errore.

INIZIALIZZAZIONI

- In assenza di inizializzazione esplicita, le variabili globali e quelle `static` sono inizializzate a zero.
- L'inizializzazione esplicita di variabili globali o `static` va fatta con espressioni costanti (una sola volta, concettualmente prima che il programma inizi); quella di variabili automatiche si può fare anche con espressioni variabili (ad ogni inizio di funzione).
- Una variabile automatica dichiarata ed inizializzata in un blocco è inizializzata ogni volta che si entra nel blocco; una variabile `static` lo è solo la prima volta che si entra nel blocco.
- Una array può essere inizializzata con una lista di inizializzatori separati da virgola e racchiusi da { e }. Se si specificano meno inizializzatori del numero di elementi i rimanenti elementi vengono inizializzati a zero sia per variabili automatiche che esterne o statiche.
- La parola chiave `const` (derivata dal C++) indica che una variabile non può essere modificata dopo l'inizializzazione:

```
const char str[5] = "cara";
str[0]='s';            /*non ammesso*/
```

La parola chiave `const` può essere impiegata in alternativa a `#define ..`

FUNZIONI

- Sono sempre esterne, perchè C non permette di definire una funzione dentro un'altra.

- La dichiarazione di funzione è valida dal punto in cui è fatta fino alla fine del file. In assenza di dichiarazione, una funzione viene implicitamente dichiarata di tipo intero alla prima apparizione nel file (nessuna assunzione sui parametri).

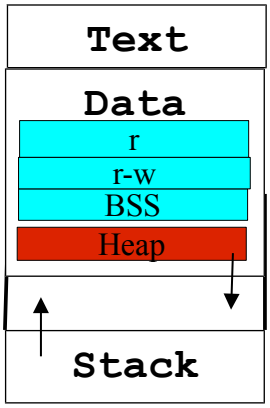
- Una inconsistenza tra dichiarazione e definizione di una funzione viene rilevata dal compilatore all'interno di uno stesso file, non in files distinti. Utile creare un header file, contenente le dichiarazioni comuni a più files, da aggiornare durante lo sviluppo del programma.

- Ad ogni chiamata di una funzione sullo *stack* viene allocata una finestra di attivazione (procedure frame o activation record) che vive durante l'esecuzione della funzione e dopo viene deallocata. Nella finestra viene memorizzato l'indirizzo di ritorno, le variabili locali, i parametri, ecc. All'inizio del programma lo stack contiene la linea di comando data alla *shell* per invocarlo; così il programma può conoscere i suoi argomenti (`argv`).

GESTIONE DELLA MEMORIA IN UNIX

Un programma in esecuzione in unix si chiama **processo**.
I processi in Unix hanno la memoria divisa in tre parti chiamate: **text segment**, **data segment** e **stack segment**.

indirizzi bassi di memoria



indirizzi alti di memoria

Il **text segment** contiene il codice del programma; è sempre di tipo read-only (non si automodifica).

Il **data segment** contiene le variabili globali e statiche. È diviso in due parti, una delle variabili inizializzate esplicitamente (che può essere a sua volta divisa in una parte read-only ed una read-write) ed una (chiamata BSS - Block Started by Symbol) delle variabili inizializzate a zero oppure non inizializzate. Poi c'è una *heap* che serve per fare *dynamic memory allocation: s*e un programma richiede o rilascia memoria durante la sua esecuzione (**malloc, calloc, free**) questo segmento aumenta o diminuisce (heap pointer).

Lo **stack segment** all'inizio del programma contiene le variabili di ambiente e la linea di comando data alla shell per invocarlo; così il programma può conoscere i suoi argomenti (**`argv`**), se ne ha. Gli indirizzi di ritorno da una funzione, i parametri, le variabili **auto**matiche (cioè locali in una funzione) vengono generalmente allocate qui. Lo stack può crescere o calare (stack pointer).

Tra il data segment e lo stack c'è uno spazio di indirizzamento inutilizzato. Lo stack cresce in questo spazio automaticamente, di quanto è necessario per soddisfare le chiamate di funzioni, mentre l'espansione del data segment viene eseguito utilizzando chiamate di sistema.

Due processi possono eventualmente condividere il text segment ma mai il data o stack segment.

IL C PREPROCESSORE

Il preprocessore del linguaggio C è un programma a sè stante, che viene eseguito prima del compilatore.

Tutte le direttive per il preprocessore iniziano con # .
Le funzioni principali offerte dal preprocessore sono le seguenti.

Per includere files

`#include<filename>` la ricerca avviene in directory dipendenti dal sistema

`#include"filename"` la ricerca avviene nella directory del file da compilare (si usa per includere header files di utente o files contenenti diverse componenti di un programma su più files)

Per definire costanti simboliche

```
#define   MAXDIM        10
#define   TIPO          int
```

Per definire macros

```
#define   forever       for(;;)

#define   max(A,B)      ((A)>(B)) ? (A): (B)
          e poi x = max(p+q , r+2);

#define   mult_per_2(a)  ((a)+(a))
          e poi x = mult_per_2(5);
```

I parametri di una macro non sono variabili: non è definito il loro tipo, nè viene loro assegnata memoria ⇒ non sono in conflitto con variabili esistenti con lo stesso nome.

Le macro vengono normalmente eseguite più velocemente delle funzioni, perché non è necessario il salvataggio degli argomenti sullo stack.

Dal punto di vista operativo l'utilizzo di una macro o di una funzione non produce un risultato equivalente:

```
#define   mult_per_2(a)  ((a)+(a))
int mult_by_to(int a) {return a+a;}
```

... infatti sul parametro della macro non viene eseguito alcun controllo di tipo (**`mult_per_2`** può ricevere un parametro a di tipo qualsiasi); la funzione (nell'esempio), invece, presuppone un argomento intero e restituisce un valore intero. Se alla funzione viene passata una costante reale, il compilatore può comportarsi diversamente, in presenza del prototipo (in assenza di prototipo il risultato è imprevedibile).

Costanti simboliche e macros possono essere **#undef**, così la loro visibilità è da dove vengono definite fino a quando vengono **#undef o** alla fine del file.

Le funzioni della libreria standard talvolta sono **#define** come macros basate su altre funzioni di libreria. Ad esempio una macro comunemente definita in stdio.h è la seguente:

```
#define   getchar()  getc(stdin)
```

Per compilazione condizionata

Le compilazioni condizionali sono particolarmente utili nella fase di debugging, durante lo sviluppo di un programma, per attivare o disattivare porzioni di codice.

```
#if !defined (NULL)
     #define NULL   0
#endif


#if 0
     non compilare questo pezzo di codice
#endif

#if !defined (HDR)                (anche #ifndef HDR)
     #if   SYSTEM==SYSV
           #define HDR   "sysv.h"
     #elif SYSTEM==BSD
           #define HDR   "bsd.h"
     #elif SYSTEM==MSDOS
           #define HDR   "msdos.h"
     #else
           #define HDR   "default.h"
     #endif
#endif
#include HDR
```