SPAZIO DI INDIRIZZAMENTO VIRTUALE

Manuale linguaggio C

Allocazione della memoria per i programmi C

Il C supporta **tre** modalità di allocazione della memoria: l'allocazione statica, l'allocazione automatica e l'allocazione dinamica.

- L'allocazione **automatica** è quella che avviene per gli argomenti di una funzione e per le sue variabili locali (le cosiddette **variabili automatiche**), che esistono solo per la durata della funzione (***main*** incluso). Lo spazio per queste variabili viene allocato nello **stack** quando viene eseguita la funzione (a **run-time**) e liberato quando si esce dalla medesima. La dimensione non è modificabile a run-time (è nota al "compile-time")

- L'allocazione **statica** è quella con cui sono memorizzate le variabili **globali** e le variabili **statiche** (le variabili il cui valore deve essere mantenuto per tutta la durata del programma e che vengono allocate nel segmento dei **dati** all'avvio del programma e lo spazio da loro occupato non viene liberato fino alla sua conclusione). Il blocco di memoria viene allocato a compile-time. La dimensione non è modificabile al run-time (è nota al "compile-time")

Allocazione della memoria per i programmi C

- L'allocazione **dinamica** della memoria, non è prevista direttamente all'interno del linguaggio C, ma risulta necessaria quando il quantitativo di memoria che serve è determinabile solo durante il corso dell'esecuzione del programma. Per questo le **librerie del C** forniscono una serie opportuna di funzioni per eseguire l'allocazione dinamica di memoria (in genere nello **heap**). Il blocco di memoria viene allocato a **run-time**. La dimensione è indicata alle funzioni ad ogni richiesta (più grande è il blocco richiesto, più tempo ci mette la funzione a riservarlo). Il blocco è rilasciabile (per poter essere utilizzato per altre allocazioni dinamiche) solo **esplicitamente** per mezzo di opportune funzioni (non è automatico)

Allocazione della memoria

A ciascun programma utente viene quindi assegnata una porzione di memoria che è automaticamente divisa nelle quattro porzioni:

- area del codice
- area dati globali e statici
- area heap
- area stack

Le dimensioni delle prime due aree (programma e dati) sono fisse e sono decise in fase di compilazione. Infatti tali dimensioni corrispondono al numero di righe comando del programma e al numero di variabili statiche definite in esso.

Le altre due aree (**heap** e **stack**) non hanno dimensione fissa, ma l'area complessiva (pari alla somma delle due) è anch'essa fissa e decisa in fase di compilazione.

Il fatto che l'heap e lo stack non abbiano dimensione fissa significa che tale dimensione varia durante l'esecuzione a seconda dell'utilizzo delle due aree.

In particolare, l'area heap **crescerà** quando verranno allocate variabili dinamiche. La sua dimensione si **ridurrà**, invece, quando una o più variabili dinamiche verranno deallocate.

Allocazione della memoria

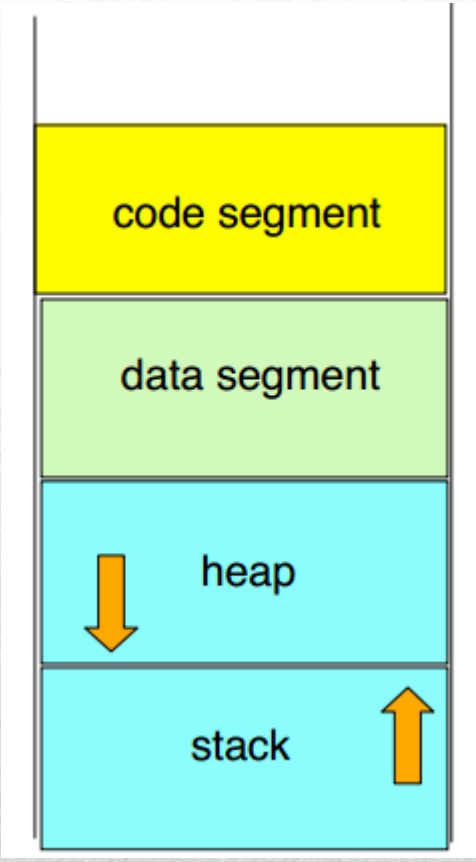
- A una variabile o struttura dati **statica** il linker assegna un indirizzo immutabile.
- Una struttura dati **dinamica** è allocata durante il funzionamento del programma, per esempio chiamando la funzione malloc() e accedendo allo spazio così ottenuto tramite un puntatore. Una variabile dinamica risiede in memoria che viene richiesta al sistema operativo durante il funzionamento del programma. Al momento dell'allocazione non si possono fare assunzioni sul contenuto di tale memoria: potrebbe essere azzerata ma potrebbe contenere informazioni residue di precedenti allocazioni poi liberate. Ad ogni malloc() deve corrispondere una free(), in mancanza della quale abbiamo una situazione di perdita di memoria (**memory leakage**) e la dimensione del programma in esecuzione aumenterà in continuazione.
- Una variabile cosiddetta **automatica** viene allocata sullo stack e scompare al termine del blocco di codice che la dichiara.

Organizzazione della memoria

Code segment: contiene il codice eseguibile del programma
Data segment: contiene le variabili globali
Heap: contiene le variabili dinamiche
Stack: contiene i record di attivazione

Code segment + data segment = dimensione fissata staticamente (a tempo di compilazione)

La dimensione dell'area associata a **stack + heap** è fissata staticamente: mano a mano che lo stack cresce, diminuisce l'area a disposizione dell'heap, e viceversa.



Le classi di memorizzazione

La **classe di memorizzazione** di una variabile è individuata da due caratteristiche: la **visibilità** e la **durata** della variabile.

L'ambito di visibilità è il termine tecnico che denota **la parte del testo sorgente C in cui è attiva la dichiarazione di un nome**.

Nel linguaggio C, viene offerta la possibilità di condividere variabili e di delimitare le porzioni di codice che sfruttano tali condivisioni, mediante la definizione dell'ambito di visibilità, o **scope**, delle variabili.

Inoltre, le variabili hanno una **durata**, che descrive il lasso temporale di memorizzazione dei valori di una variabile:

- nel caso di variabili con **durata fissa**, i valori memorizzati vengono mantenuti anche all'esterno dell'ambito di visibilità

Le classi di memorizzazione

Esempio

```
void func()
{
  int j;
  static int ar[]={1,2,3,4}
        … … …
}
```

Le variabili j ed ar hanno entrambe **visibilità a livello di blocco**, perché definite all'interno di un blocco: possono essere referenziate solo dalle istruzioni che appartengono al blocco (il corpo della funzione func())

Le variabili j ed ar sono dette **locali**

static: durata di memorizzazione statica invece che automatica; la variabile possiede una locazione di **memoria permanente** e quindi può mantenere il suo valore durante tutta l'esecuzione del programma

La variabile j ha **durata automatica**, mentre ar ha **durata fissa**, perché dichiarata **static**:

- A j viene automaticamente allocata memoria ogni volta che viene eseguito il blocco che la contiene (può avere indirizzi diversi per esecuzioni diverse del blocco di codice)
- ar viene allocata la prima volta che viene eseguito il blocco e mantiene l'indirizzo originale per l'intera esecuzione del programma

Durata fissa e durata automatica

Le variabili con **durata fissa** sono permanenti, mentre le variabili con **durata automatica** sono allocate più volte durante l'esecuzione del programma

- Ad una variabile fissa viene associata una locazione di memoria all'inizio del programma, che non cambia fino al termine dello stesso
- Ad una variabile automatica viene allocata memoria ogni volta che si entra nel suo ambito di visibilità; se il codice che appartiene all'ambito di visibilità della variabile viene rieseguito, la variabile viene generalmente **allocata altrove**: non si mantiene il valore della variabile fra due esecuzioni successive

Le variabili locali sono automatiche per default, ma possono essere rese fisse se dichiarate **static**

La parola chiave **auto** definisce esplicitamente una variabile automatica, ma è usata raramente perché ridondante

Inizializzazione delle variabili

Le variabili fisse vengono inizializzate una sola volta, mentre quelle automatiche vengono inizializzate ogni volta che viene eseguito il blocco che le contiene

```
void increment()
{
  int j=1;
  static int k=1;

  j++;
  k++;
  printf("j: %d\t k: %d\n", j, k);
}

main()
{
  increment();
  increment();
  increment();
}
```

La funzione **increment()** incrementa i valori delle variabili j e k, entrambe inizializzate a 1

Il risultato dell'esecuzione del programma è:

j: 2 k: 2

j: 2 k: 3

j: 2 k: 4

Uso di variabili con durata fissa

```
#define ODD  1
#define EVEN 0

void print_header(char *chap_title)
{
    static char page_type=ODD;

    if (page_type == ODD)
    {
       printf("\t\t\t\t\t%s\n\n", chap_title);
       page_type = EVEN;
    }
    else
    {
       printf("%s\n\n", chap_title);
       page_type = ODD;
    }
}
```

Le variabili con durata fissa sono comunemente impiegate per tenere traccia del numero di volte che una funzione viene eseguita e per modificarne il comportamento ad intervalli regolari

La variabile **page_type** agisce da elemento di controllo: quando il numero della pagina è dispari, la funzione stampa la stringa puntata da **chap_title** sul lato destro della pagina; se il numero della pagina è pari, la stringa appare spostata a sinistra

page_type deve essere fissa, altrimenti si stamperebbe sempre l'intestazione relativa alle pagine dispari

Ambito di visibilità delle variabili

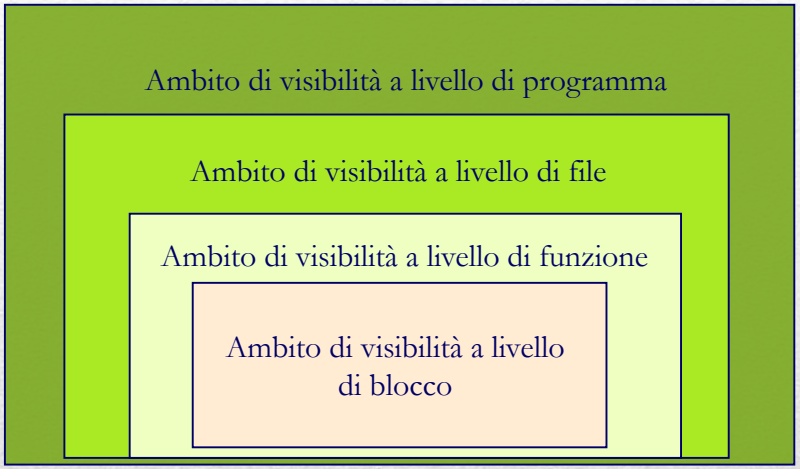
L'**ambito di visibilità** di una variabile definisce la regione di codice da cui è possibile accedere alla variabile

Esistono quattro tipi di ambiti di visibilità: **programma**, **file**, **funzione** e **blocco**

- L'ambito di visibilità a livello di **programma** implica che una variabile è accessibile da tutti i file sorgente; le variabili con ambito di visibilità a livello di programma sono **variabili globali**
- L'ambito di visibilità a livello di **file** implica che una variabile è accessibile dal punto in cui è dichiarata fino alla fine del file sorgente in cui si trova
- L'ambito di visibilità a livello di **funzione** implica che una variabile è accessibile dall'inizio alla fine della funzione in cui è dichiarata; le variabili con ambito di visibilità a livello di funzione sono **variabili locali alla funzione**
- L'ambito di visibilità a livello di **blocco** implica che una variabile è accessibile dal punto in cui è dichiarata fino alla fine del blocco in cui si trova; le variabili con ambito di visibilità a livello di blocco sono **variabili locali al blocco**

Ambito di visibilità delle variabili

L'ambito di visibilità di una variabile è determinato dalla posizione della dichiarazione:



Le relazioni gerarchiche tra gli ambiti di visibilità

Le variabili dichiarate all'interno di un blocco hanno ambito di visibilità a livello di blocco

Le variabili dichiarate all'esterno di un blocco hanno ambito di visibilità a livello di file se sono **static**,

Ambito di visibilità delle variabili

Esempio:

```
int i;          /* ambito di visibilità a livello di programma */
static int j;   /* ambito di visibilità a livello di file */

int func(k)     /* ambito di visibilità a livello di funzione */
{
  int l;        /* ambito di visibilità a livello di funzione */
  …
  {
    int m;      /* ambito di visibilità a livello di blocco */
    … … …
  }
  …
}
```

I parametri delle funzioni hanno ambito di visibilità a livello di funzione: sono trattati come se fossero la prima dichiarazione di variabile nel blocco di livello più alto contenuto nella funzione

Ambito di visibilità delle variabili

Il C consente l'attribuzione dello stesso nome a variabili diverse con ambiti di visibilità distinti

È anche possibile che variabili con lo stesso nome abbiano ambiti parzialmente sovrapposti: la variabile con l'ambito di visibilità più limitato preclude temporaneamente la visibilità dell'altra

```
int j=10; /* visibilità a livello di programma */
main()
{
  int j;   /* visibilità a livello di blocco:
              * nasconde la variabile globale j */
  for (j=0; j<5; j++)
      printf("j: %d", j);
}
```

L'esecuzione del programma produce:

j: 0
j: 1
j: 2
j: 3
j: 4

La variabile globale **j** mantiene inalterato il valore iniziale 10

Ambito di visibilità a livello di blocco e funzione

Una variabile con ambito di visibilità a livello di blocco **non può essere acceduta** dall'esterno del blocco in cui è dichiarata

- consente di proteggere la variabile da effetti collaterali non desiderati
- riduce la complessità del programma, rendendolo più leggibile e mantenibile

Ambito di visibilità a livello di file e programma

Associare un ambito di visibilità a livello di file a una variabile significa renderla referenziabile nella parte rimanente del file in cui è definita

- Se il file contiene più funzioni, tutte le funzioni che seguono la dichiarazione della variabile sono in grado di referenziarla

Per dichiarare una variabile con ambito di visibilità a livello di file occorre inserire la dichiarazione al di fuori delle funzioni e usare la parola chiave **static**

Le variabili con ambito di visibilità a livello di programma, dette **variabili globali**, sono visibili in tutti i file sorgente (compreso quello in cui vengono dichiarate)

Una variabile globale deve essere dichiarata al di fuori delle funzioni e **non** usando la parola chiave **static**

I due significati di static

All'interno di un blocco, **static** attribuisce ad una variabile durata fissa, anziché automatica

All'esterno di una funzione, **static** non è correlata alla durata della variabile, ma ne controlla l'ambito di visibilità a livello di file, anziché di programma

La parola chiave **static** specifica sia l'ambito di visibilità che la durata di una variabile:

- All'interno di un blocco, le regole di visibilità del blocco sono più stringenti di quelle a livello di file: la durata fissa è l'unico effetto che si manifesta
- All'esterno di una funzione, la durata è già fissa: l'ambito di visibilità a livello di file è l'unico effetto che si manifesta

Esempio

```
static int cont = 3;

int nextValue(){
        return cont++;
}
```

La variabile cont è inaccessibile fuori da questo file (il suo scope di definizione); l'unico modo di accedervi è tramite la funzione nextValue

Se un altro file definisse un'altra variabile globale *cont*, non ci sarebbe comunque collisione, perché la prima esternamente "non esiste".

```
int nextPrime(void) {
  static lastprime = 2;
  do
     lastprime++;
  while (!isPrime(lastprime));
  return lastprime;
}
```

Le variabili globali

L'uso delle **variabili globali** dovrebbe essere limitato perché aumenta la complessità dei programmi e li rende difficilmente mantenibili

Le variabili globali possono portare a conflitti tra funzioni se, per errore, vengono scelti nomi uguali per variabili globali distinte

Quando occorre condividere dati tra funzioni diverse, è buona regola passare i dati come parametri o passare puntatori all'area di memoria condivisa

Definizioni e allusioni

Finora, la dichiarazione di variabile corrispondeva all'allocazione di memoria per quella variabile: l'allocazione di memoria è, in realtà, il risultato di un solo tipo di dichiarazione, detta **definizione**

Le variabili globali consentono un secondo tipo di dichiarazione, detta **allusione**: non si alloca memoria, ma si informa il compilatore che esiste una variabile del tipo specificato definita altrove

```
main()
{
    extern int f();                 /* allusione a funzione */
    extern int j;                   /* allusione a variabile */
    extern float f_array_of_f[];    /* allusione a variabile */
    … … …
```

Definizioni e allusioni

Le variabili globali seguono le stesse regole delle funzioni: ogni volta che si utilizzano variabili definite in altro file, è necessario dichiararle con allusioni

La parola chiave **extern** specifica che la variabile è definita altrove

Le allusioni consentono al compilatore di effettuare i controlli di tipo: per ogni variabile globale, possono essere presenti un numero qualunque di allusioni, ma una sola definizione

Lo specificatore register

La parola chiave **register** consente di suggerire al compilatore quali variabili dovrebbero essere memorizzate nei registri

Il livello di supporto offerto dai compilatori allo specificatore **register** è molto variabile: alcuni compilatori memorizzano effettivamente 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

Sono candidati ideali per la memorizzazione **register** i contatori dei cicli **for**, sui quali vengono effettuate molte operazioni, temporalmente vicine

```
#include <stdio.h>

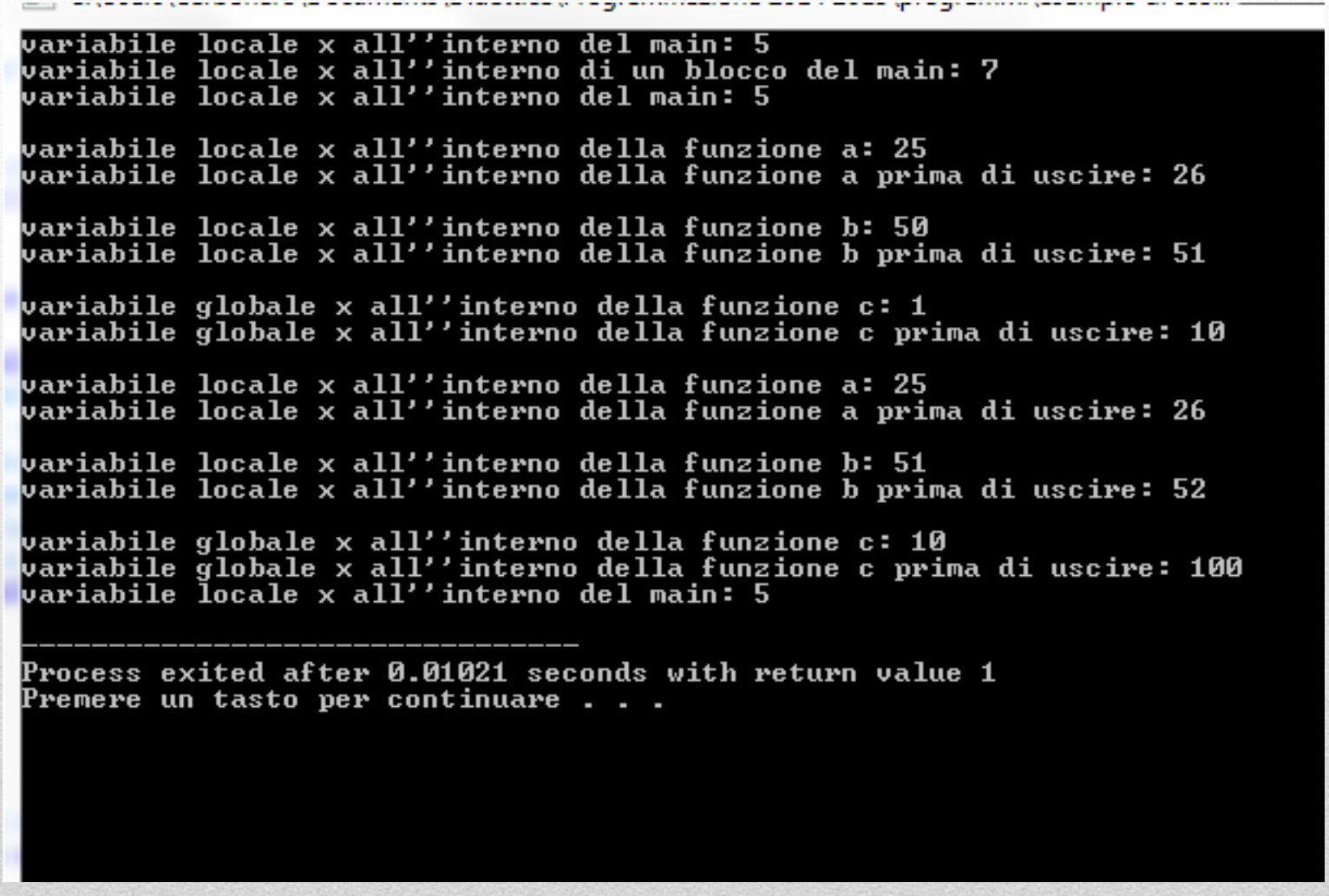
void a (void);
void b (void);
void c (void);

int x=1;              // variabile globale
int main()
{
  int x=5;            //variabile locale al main() con lo stesso nome di una globale
  printf("variabile locale x all'interno del main: %d\n", x);
  {
    int x=7;
    printf("variabile locale x all'interno di un blocco del main: %d\n", x);
  }
  printf("variabile locale x all'interno del main: %d\n", x);
  a();                //a ha variabile automatica locale x
  b();                //b ha variabile locale statica c
  c();                //c usa variabile globale x
  a();                //a re-inizializza la variabile locale automatica x
  b();                //la variabile statica locale x mantiene il suo precedente valore
  c();                //la variabile globale x mantiene il suo valore
  printf("variabile locale x all'interno del main: %d\n", x);
  return 1;
}
```

```
void a(void)
{
  int x=25;          //inizializzata ogni volta che viene richiamata
  printf("variabile locale x all'interno della funzione a: %d\n", x);
  x++;
  printf("variabile locale x all'interno della funzione a prima di uscire: %d\n", x);
}

void b(void)
{
  static int x=50;  //inizializzata solo la prima volta che viene richiamata
  printf("variabile locale x all'interno della funzione b: %d\n", x);
  x++;
  printf("variabile locale x all'interno della funzione b prima di uscire: %d\n", x);
}

void c(void)
{
  printf("variabile globale x all'interno della funzione c: %d\n", x);
  x*=10;
  printf("variabile globale x all'interno della funzione c prima di uscire: %d\n", x);
}
```


variabile locale x all''interno del main: 5
variabile locale x all''interno di un blocco del main: 7
variabile locale x all''interno del main: 5
variabile locale x all''interno della funzione a: 25
variabile locale x all''interno della funzione a prima di uscire: 26
variabile locale x all''interno della funzione b: 50
variabile locale x all''interno della funzione b prima di uscire: 51
variabile globale x all''interno della funzione c: 1
variabile globale x all''interno della funzione c prima di uscire: 10
variabile locale x all''interno della funzione a: 25
variabile locale x all''interno della funzione a prima di uscire: 26
variabile locale x all''interno della funzione b: 51
variabile locale x all''interno della funzione b prima di uscire: 52
variabile globale x all''interno della funzione c: 10
variabile globale x all''interno della funzione c prima di uscire: 100
variabile locale x all''interno del main: 5
Process exited after 0.01021 seconds with return value 1
Premere un tasto per continuare . . .