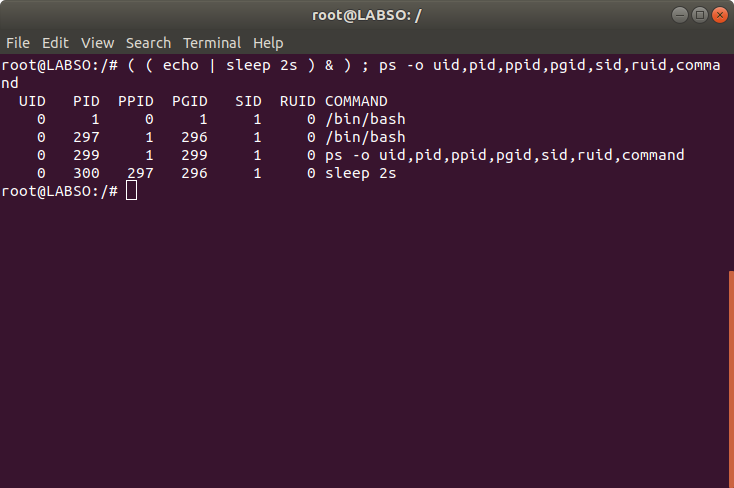
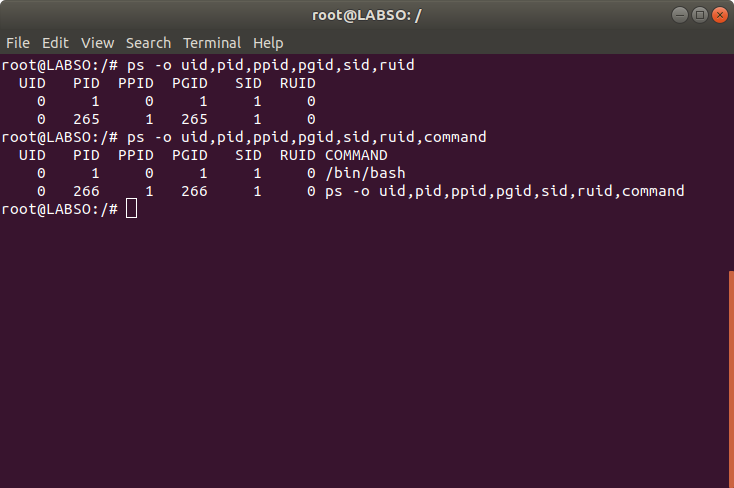
**Laboratorio Sistemi Operativi: Process Vars**

* Un **processo è un’istanza in esecuzione di un programma**.
* Quando eseguo un programma, viene creata **un’immagine nella memoria** che contiene una rappresentazione dei contenuti del codice binario e **quest’istanza** è quella che chiamiamo **processo**.
* All’interno del sistema, **ogni processo** è caratterizzato da una serie di attributi, in particolare **contiene**:
  + **Codice**: Il **codice che viene eseguito** (codice che dipende dall’architettura, CPU e dall’ambiente d’esecuzione in cui ci troviamo).
  + **Stato**:
    - Composto da una serie di elementi:
      * **Program Counter**: cursore di posizione che specifica, all’interno dell’istanza (che è costituita da tante istruzioni) il punto in cui ci troviamo in un determinato momento (fa da segnalibro per l’istruzione da eseguire).
      * **Stack**: utile per registrare il punto di ritorno in caso di chiamata di funzione.
  + **Dati**: Dove **sono salvati i valori di alcune variabili** settate durante l’esecuzione.
* Programmi e processi sono due cose differenti:
  + **Programma**:
    - **Entità passiva** costituita tipicamente da uno o più file, rappresentanti del codice di programmazione da eseguire.
  + **Processo**:
    - È un’istanza di un programma, **un’entità attiva ed in esecuzione**.
* Un **processo è identificato** da:
  + UID:
    - **User ID**
    - ID che **identifica l’user** che ha generato il processo.
  + PID:
    - **Process ID**
    - **Identificatore unico che rappresenta il processo** in memoria (unico durante l’esecuzione).
    - In ogni istante temporale, **c’è un unico processo associato a quell’identificativo**.
  + PPID:
    - **Parent Process ID**
    - È un **identificativo** di processo ma fa **riferimento al “genitore”**, cioè al processo che ha creato il processo.
    - Tendenzialmente questo ID **non cambia durante l’esecuzione**, ma può capitare che il padre termini la sua esecuzione prima del suo processo figlio:
      * In questo caso il **processo figlio diventa un processo orfano**.
      * Visto che tutti i processi figli devono avere un padre, questo **processo orfano viene “adottato” dal processo init** (processo principale).
  + PGID:
    - **Process Group ID**
    - **Identificatore di gruppo**, ci permette di identificare e gestire un insieme di processi.



* + - * I comandi **echo** e **sleep** vengono eseguiti in **un altro processo bash in background**.
      * Questi due comandi generano **due processi differenti** in memoria, ma hanno stesso PGID.
      * Il comando **ps** viene eseguito sulla **bash** attualmente in esecuzione e mostra lo stato dei processi.
  + SID:
    - **Process Session ID**
    - Una **sessione è un insieme di gruppi di processi**, in particolare **ogni sessione è collegata al massimo con un terminale** (può non averlo) e viceversa, **ogni terminale è collegato ad una sessione**.
    - Se chiudo il terminale, il sistema chiude tutti i processi che erano collegati a quel terminale usando il SID associato.
  + RUID:
    - **Real User ID**
    - Può succedere che un **processo venga invocato da un utente ma eseguito come un altro**.
    - Ad esempio, quando si esegue un comando ma questo ha bisogno di privilegi root perché modifica qualche file di sistema, in quel caso il **RUID differisce dal UID**.
* **Comando ps**:
  + Il comando **ps** mostra **informazioni relative ai processi attualmente in memoria** nel sistema.
  + Invocando **ps** con il **flag -o [option]** è possibile **visualizzare** sul terminale **alcune particolari etichette** dei processi:
    - **ps -o uid, pid, ppid, pgid, sid, ruid**:
      * vengono **visualizzati i processi principali attivi**, mostrando a video le loro etichette (in questo caso sono i **vari ID che li caratterizzano**).
    - **ps -o uid, pid, ppid, pgid, sid, ruid, command**:
      * esecuzione analoga a quella descritta precedentemente, solo che oltre ai vari ID, viene **stampato** anche il **nome del processo** (il comando eseguito).



* **Daemon**:
  + I **processi daemon** (demoni) sono una categoria specifica di **processi particolari**.
  + Solitamente si usa questo termine per riferirsi a **processi in esecuzione in background** tipicamente **senza un terminale collegato**.
  + **Creazione** di un demone:
    - **Processo P** (parent) **crea processo** figlio **C** (children).
    - **Processo P** **termina** la sua esecuzione **prima** del figlio C.
    - **Processo C** rimane in **esecuzione** ed esegue la funzione **setsid()**, funzione che avvia una nuova sessione senza un terminale collegato ad essa.
    - **Processo C** crea un **nuovo processo** figlio **G** (grandchild).
    - **Processo C** **termina** la sua esecuzione **prima** del figlio G.
    - Il **processo G** è un **daemon**.
* **Memoria**:
  + La memoria ha una sua organizzazione (può variare da sistema a sistema):
    - **OS Kernel**: il sistema riserva gli indirizzi di memoria più bassi per il Kernel.
    - **Text Segment**: riservato per le **istruzioni eseguite dalla CPU**.
    - **Data Segment**: riservato per le **inizializzazioni** (int a = 5).
    - **BSS Segment**: riservato per le **variabili non inizializzate** (int b).
    - **Heap**: spazio di memoria con **gestione dinamica** personalizzata dell’allocazione.
    - **Stack**: spazio di memoria con **gestione automatica dell’allocazione**, come ad esempio il punto di ritorno delle chiamate di funzione.
  + **Stack** e **Heap** si **contendono lo spazio** (se Heap cresce lo Stack diminuisce e viceversa).
  + In C la gestione della memoria avviene in **modo diretto**, non ci sono funzioni ad alto livello (salvo l’uso di librerie), ma di fatto **è demandato al programmatore la gestione** ottimale delle richieste e rilasci della memoria.
  + Non c’è un **garbage collector**, per cui **se alloco memoria devo anche liberarla**, altrimenti le zone di memoria allocate ma non usate verranno mantenute per tutta la durata dell’esecuzione del mio processo (spreco di memoria).
  + Le variabili possono essere:
    - **Statiche**:
      * Sono **variabili che vengono inizializzate al momento della dichiarazione** (salvate nel **Data Segment**) **o non inizializzate**, viene dichiarato solo il tipo (salvate nel **BSS Segment**).
    - **Dinamiche**:
      * Sono variabili che **contengono un riferimento ad una cella di memoria** (puntatori).
      * Sono **settate a run-time** e se non sono inizializzate assumo un valore casuale.
      * Devono essere **gestite** **mediante** apposite funzioni:
        + **malloc():**

Utilizzando la funzione malloc() **posso allocare e riservare una certa porzione di memoria**.

Questa zona di memoria verrà **referenziata mediante una variabile di tipo puntatore**.

Non è detto che utilizzando la malloc(), mi venga data una zona di memoria vuota, può darsi che **contenga dati vecchi**.

* + - * + **free():**

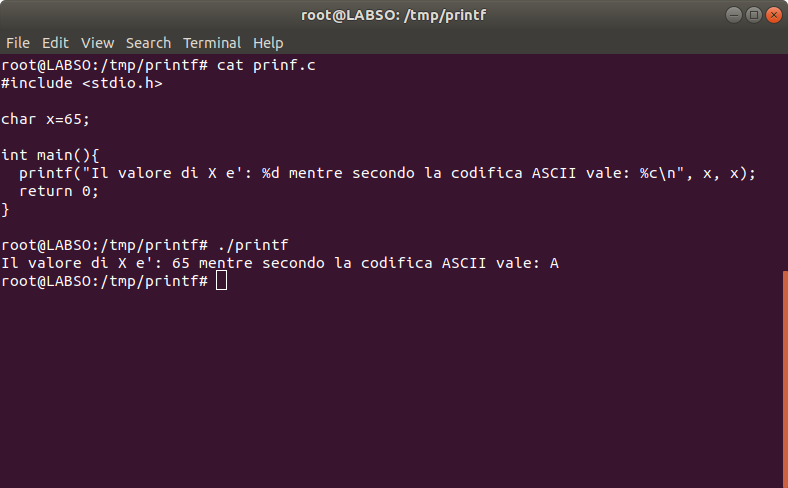
Una volta utilizzata la **zona di memoria** appena allocata, **devo liberarla**.

La funzione **free() dealloca zone di memoria**, in modo da non sprecare risorse.

* + - **Automatiche**:
      * **Variabili create localmente all’interno di funzioni** (se non inizializzate, assumono valori casuali).
* **Linguaggio C:**
  + Nel linguaggio C, si hanno pochi e semplici tipi di dato (per strutture più complicate, dobbiamo usare costrutti e crearle da noi):
    - **Integers:**
      * **Char**:
        + **Range**: 0-255 (sottoinsieme di numeri interi, ha un range limitato)
        + **Segnaposto**:

%i: stampata come numero intero

%c: stampata secondo la rappresentazione ASCII (65 = ‘A’)



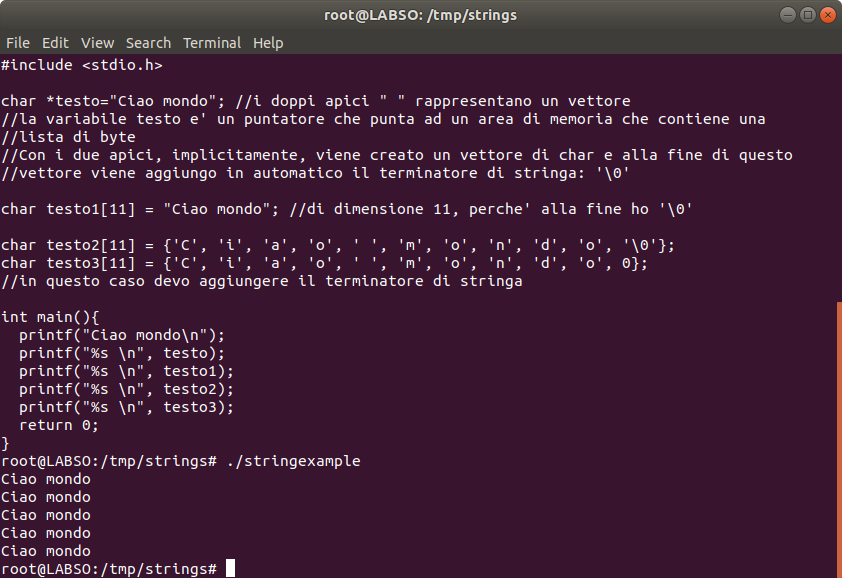
* + - * **Int**:
        + **Range**: (-32767 : +32767)
        + **Segnaposto**: %i o %d
      * **Long Int**:
        + **Range**: (-2147483647 : +2147483647)
        + **Segnaposto**: %li
    - **Boolean**:
      * **Non esistono** tipi di dato booleani in C, vengono usati valori interi:
        + **0: falso**
        + **1: vero**
      * Usando la direttiva #define, si definiscono le macro:

**#define BOOL char**

**#define FALSE 0**

**#define TRUE 1**

* + - **String**:



* + - * Le stringhe non sono un tipo di dato primitivo, sono infatti **vettori di char** (vettori di interi con range 0-255) e come **ultimo elemento dell’array**, hanno un **terminatore** di **stringa** (obbligatorio):
        + **0**: numero
        + **‘\0’**: carattere
      * Sono **simili a puntatori** a **celle di memoria** che **contengono** **char**.
    - **Pointers**:
      * Sono variabili che **fanno riferimento a delle aree di memoria**.
      * **Operatori**:
        + **&var**:

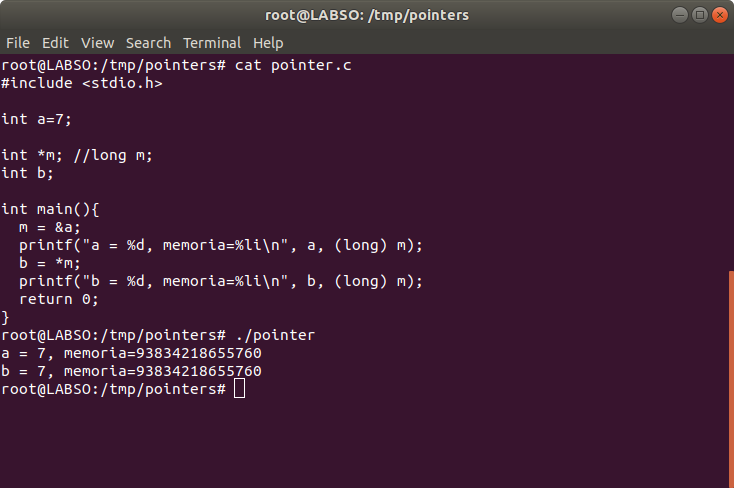
Per accedere alla locazione di memoria assegnata alla variabile var, come **indizio di memoria**.

* + - * + **\*pointer**:

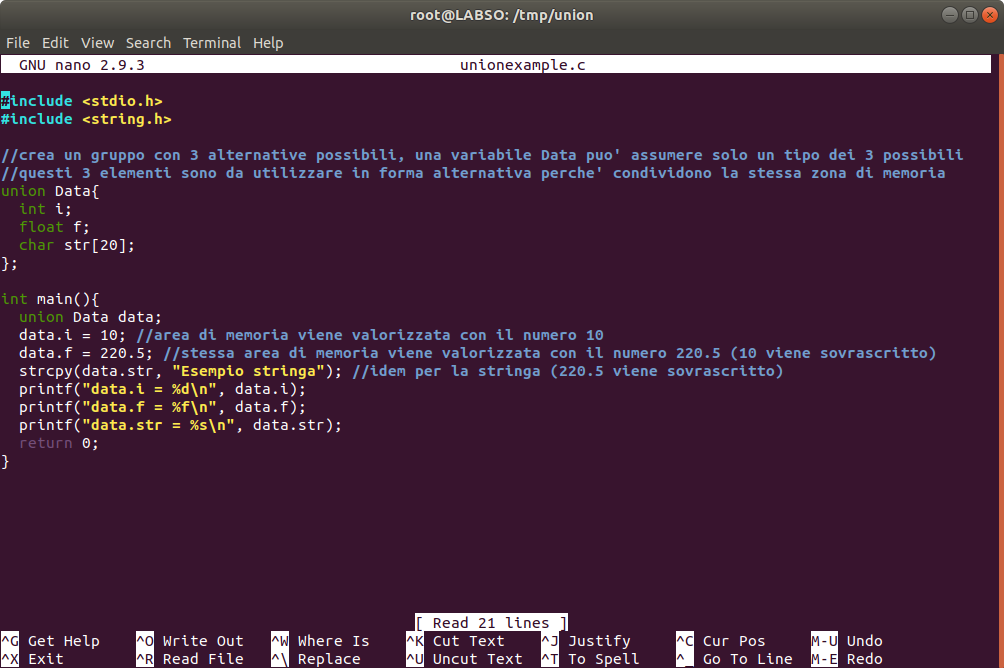
Definizione esplicita di un puntatore

Permette di **accedere al contenuto** dell’area di memoria puntata dalla variabile pointer.

* + - * **Dichiarazione**:
        + **int \*pointer** o **int\* pointer** (il primo modo è più concettualmente corretto, il puntatore si lega alla variabile, non al tipo).



* + - **Union**:
      * Permettono di **costruire delle alternative alle variabili**.
      * Sono strutturalmente simili alle struct, **creano un’aggregazione di variabili**, ma differiscono per l’accesso a queste variabili.
      * I **campi contenuti** all’interno della **union** **sono** da vedere come delle **alternative** (non si possono usare insieme né contemporaneamente), in quanto **condividono la stessa zona di memoria**.



* + - **Struct**:
      * Permette di costruire degli **aggregati di variabili** (definite come campi).
      * I **campi delle struct** sono **accessibili** mediante la **dot notation**.
      * Tipicamente struct e typedef vengono usati per maggiore **leggibilità** del codice.
    - **Typedef**:
      * È un costrutto **utilizzato per rinominare** dei tipi di dato (è una macro dedicata per i tipi)
      * Viene usata per **creare degli alias**, i tipi di dato complessi o complicati, vengono rinominati per avere delle diciture più semplici ed auto-esplicative.
    - **Puntatori/Array:**
      * **Puntatore**: <type> \*<var>
      * **Array**:
        + Sono dei **puntatori ad aree di memoria** che contengono dati omogenei.
        + <type> <var> [<size>]
      * **\*(a+i) = a[i] = i[a]**