

Linguaggio C: approfondimento

Pt. 2

Reti di Calcolatori A.A. 2023/24

Prof.ssa Chiara Petrioli - Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Michele Mastrogiovanni - Dipartimento di Ingegneria Informatica

1. Stringhe C

Manipolazione di stringhe C

3

Stringhe C: array di `char` terminati dal carattere nullo `'\0'`

```
1 char greet[6] = "hi";
```



- Format string `%s` permette l'uso di `printf/scanf` di stringhe
- **Input:** suggerito l'uso della funzione `fgets` – non lascia caratteri nel buffer

Come fare per

- Assegnare il valore di una stringa (non durante inizializzazione)?
- Confrontare stringhe?
- Tagliare, cercare sotto-stringhe, ... ?

```
1 char greet[6] = "hi";  
2 greet = "ciao!";
```

```
1 char greet1[6] = "ciao", greet2[6] = "hi";  
2 if(greet1 == greet2){ /*...*/ }
```

Libreria `string.h`

Il linguaggio C fornisce una libreria per la manipolazione di stringhe: `string.h`. Per usare le funzioni disponibili, bisogna importare la libreria tramite l'apposita direttiva al preprocessore `#include`.

Alcune funzioni incluse:

➤ `strlen(const char *str)`

Restituisce la lunghezza di una stringa, ovvero il numero di caratteri prima del *primo* `'\0'`

```
1 char greet[6] = "he\0y";  
2 printf("%lu\n", strlen(greet));
```

Output: 2

➤ `strncpy(char *dst, const char *src, size_t count)`

Equivalente all'assegnamento per stringhe; copia la stringa `src` nella stringa `dst` *fino al* `'\0'`, comunque *non eccedendo* `count` nel numero di caratteri copiati: *può evitare buffer overflow!*

```
1 char greet[6] = "he\0y";  
2 strncpy(greet, "ciaoooo", sizeof(greet));
```

greet non termina per `'\0'`

`strncpy` può comunque dare *problemi*: così si evita l'overflow durante la copia, ma la stringa potrebbe *non terminare* per `'\0'` !

Libreria string.h

5

➤ `strncat(char *dst, const char *src, size_t count)`

Concatena **fino a count caratteri** della stringa `src` oppure **fino al '\0'** della stessa alla stringa `dst`: *può evitare buffer overflow!*

```
1 char greet[15] = "hello";
2 char noun[6] = "world";
3 strncat(greet, noun, sizeof(greet) - strlen(greet));
```

Spazio per '\0' ?

Calcolare `count` in questo modo può dare luogo ad **unsigned underflow** e quindi eventualmente a **buffer overflow!**

➤ `strcmp(const char *strA, const char *strB)`

Confronta due stringhe **terminate da '\0'** in modo *lessicografico*; restituisce 0 se `strA` è uguale a `strB`, un valore < 0 se `strA` è minore di `strB`, ed un valore > 0 se viceversa

Nel **confronto lessicografico** le stringhe sono iterate carattere per carattere; non appena c'è una differenza, i caratteri vengono interpretati **unsigned char** e restituita la differenza tra i due

Ordinamento: '\0' < '9' < 'A' < 'Z' < 'a' < 'z'

Esiste anche la versione `strncmp` per il confronto dei **primi n** caratteri: utile se una delle due stringhe potrebbe non essere terminata da '\0'

Libreria string.h

6

- `strstr(const char *str, const char *sub)`
Trova la **prima occorrenza** della stringa `sub` all'interno della stringa `str`, **restituendone il puntatore dell'inizio** di `sub` in `str` (se presente) o `NULL` altrimenti; entrambe le stringhe **devono** terminare per `'\0'`

```
1 char doc[15] = "hello locals!";
2 char word[5] = "lo";
3 char *start = strstr(doc, word);
4 printf("Start index: %lu", start - doc);
```

Output: 3

- `strtok(char *str, const char *delim)`
Itera la stringa restituendone i *token*, ovvero sottostringhe di `str` delimitate da **uno dei caratteri** della stringa `delim`; sia `str` che `delim` devono essere terminate da `'\0'`

- Progettata per essere usata nei cicli
- `delim` è l'insieme dei caratteri delimitatori e **non il delimitatore**
- Ogni delimitatore trovato viene **sostituito con il carattere `'\0'`** all'interno di `str`!

```
1 char doc[30] = "lorem ipsum dolor sit amet";
2 int count = 0;
3 char *token = strtok(doc, " "); Prima iterazione
4 while(token){
5     count++;
6     token = strtok(NULL, " "); Successive iterazioni
7 }
8 printf("String '%s' has %d tokens", doc, count);
```

2. Dichiarazione di nuovi tipi

Dichiarazione di nuovi tipi

8

Nuovi tipi possono essere dichiarati in diversi modi:

- `struct`
Dati strutturati composti di più variabili di tipi anche diversi, salvate in **locazioni contigue** di memoria
- `union`
Collezione di variabili di tipi diversi, salvate nella **stessa locazione** di memoria
- `enum`
Tipo di dati che permette la definizione di **valori costanti** e caratterizzati da un **nome esplicito**

Dati strutturati:

struct

9

Contiene più variabili anche di tipi diversi, salvate in **locazioni contigue** di memoria.

Membri

```
1 struct Data {  
2     unsigned int day;  
3     unsigned int month;  
4     int year;  
5 } data1;  
6  
7 data1.day = 14;  
8 data1.month = 10;  
9  
10 struct Data *ptr = &data1;  
11 ptr->year = 1492;
```

Definizione di una struct Data e dichiarazione di una variabile di questo tipo, denominata data1

Variabili di tipo struct devono essere dichiarate con tipo

struct nomeStruct, ed analogamente i puntatori a variabili struct con struct nomeStruct *

L'accesso ai membri avviene in modi diversi:

- Variabile struct: operatore .
- Puntatore a struct: operatore ->

	day	month	year
data1	14	10	1492

} Layout in memoria

Dati strutturati:

struct

10

Dettagli aggiuntivi riguardo le `struct`:

- In una dichiarazione di `struct`, il nome è **opzionale** (`struct` anonima)
- Il puntatore ad una variabile `struct` può essere interpretato come **puntatore al primo membro** (e viceversa)
- La **dimensione** di una variabile `struct` è **almeno la somma** della dimensione dei propri membri
Almeno: tra le locazioni di memoria di un membro e del seguente può essere inserito un po' di **padding** da parte del compilatore, ai fini di allineamento a blocchi di memoria
- Inizializzazione di variabili `struct` effettuata in modo simile agli array

```
1 struct Data {  
2     unsigned int day;  
3     unsigned int month;  
4     int year;  
5 };  
6  
7 struct Data data1 = {14, 10, 1492};
```



```
1 struct Data {  
2     unsigned int day;  
3     unsigned int month;  
4     int year;  
5 } data1 = {14, 10, 1492};
```

Collezioni di tipi:

union

11

Collezione di variabili di tipi diversi, salvate nella **stessa locazione** di

memoria

```
1 union IdLibro {  
2     long ean;  
3     char isbn[15];  
4 };
```

Dichiarazione analoga ad una struct

```
5  
6 union IdLibro book;  
7  
8 book.ean = 9780131103627;  
9 printf("EAN: %ld\n", book.ean);  
10 printf("ISBN: %s\n", book.isbn);  
11 printf("Size: %lu\n", sizeof(book));  
12  
13 strncpy(book.isbn, "978-0131103627\0", 15);  
14 printf("EAN: %ld\n", book.ean);  
15 printf("ISBN: %s\n", book.isbn);  
16 printf("Size: %lu\n", sizeof(book));
```

Undefined

sizeof(IdLibro)

Undefined

Invariata

- Può essere valorizzato **un solo membro** alla volta
- Il contenuto di una stessa zona di memoria può essere interpretato come il tipo del membro valorizzato
- Dichiarazione variabili `union` ed accesso ai membri analogo a quello delle `struct`

Collezioni di tipi: `union`

12

Dettagli aggiuntivi riguardo le `union`:

- In una dichiarazione di `union`, il nome è **opzionale** (`union` anonima)
- Il puntatore ad una variabile `union` può essere interpretato come **puntatore ad uno qualunque dei suoi membri** (e viceversa)
- La **dimensione** di una variabile `union` è **almeno** la dimensione del **tipo più grande** tra quello dei propri membri
Almeno: come avviene nelle `struct`, il compilatore può inserire **padding** per allineamento della memoria
 - Inizializzazione di variabili `union`: se forniti valori per tutti i membri, **undefined behavior**

```
1 union IdLibro book;  
2 strncpy(book.isbn, "978-0131101630", 15);
```



```
1 union IdLibro book = {.isbn = "978-0131101630"};
```

Il membro da valorizzare può essere specificato al momento dell'inizializzazione tramite l'operatore di accesso

Perché usare le `union`?

- Utilizzo efficiente di una stessa locazione di memoria
- Implementazione di *pseudo-polimorfismo* – altrimenti impossibile in C
- Accesso diretto a memoria o hardware in programmazione embedded (es. singoli bit di registri CPU)

Leggibilità e consistenza per valori costanti: `enum`

13

Tipo di dati che permette la definizione di **valori costanti** e caratterizzati da un **nome esplicito**.

- Facilità la leggibilità e la comprensione dei programmi
- Può essere usato negli `switch-case`
- Valore **automaticamente assegnato** dal compilatore, se non specificato tramite una *constant expression*
- **Non** è possibile **riutilizzare** lo stesso identificatore in diversi `enum`

```
1 enum TrafficLight {
2     GREEN,
3     YELLOW,
4     RED
5 };
6
7 enum TrafficLight sem = GREEN;
8
9 printf("%d\n", GREEN);
10 printf("%d\n", YELLOW);
11 printf("%d\n", RED);
```

Output:

0

1

2

```
1 enum TrafficLight {
2     GREEN=10,
3     YELLOW=20,
4     RED=30
5 };
6
7 enum TrafficLight sem = GREEN;
8
9 printf("%d\n", GREEN);
10 printf("%d\n", YELLOW);
11 printf("%d\n", RED);
```

Output:

10

20

30

Migliori dichiarazioni di tipi personalizzati

14

- Le dichiarazioni di variabili di tipi personalizzati sono inutilmente prolisse

```
1 struct Data data1 = {14, 10, 1492};  
2 union IdLibro book;  
3 enum TrafficLight sem = GREEN;
```

- Allo stesso modo, anche le dichiarazioni di puntatori a funzioni potrebbero essere migliorate

```
1 int (*fnPtr)(unsigned int*, int, int*) = routine;
```

Migliori dichiarazioni di tipi personalizzati: `typedef`

15

Tramite `typedef` è possibile utilizzare un identificatore come alias di un nome di tipo, possibilmente migliorando la leggibilità del codice.

Considerando queste dichiarazioni:

```
1 struct Data {  
2   unsigned int day;  
3   unsigned int month;  
4   int year;  
5 };
```

```
1 union IdLibro {  
2   long ean;  
3   char isbn[15];  
4 };
```

```
1 enum TrafficLight {  
2   GREEN,  
3   YELLOW,  
4   RED  
5 };
```

```
1 int function(  
2   struct Data d,  
3   union IdLibro b,  
4   enum TrafficLight s  
5 );
```

Un programma potrebbe essere il seguente:

```
1 struct Data currDate = {14, 10, 1492};  
2 union IdLibro bookC = {9780131101630};  
3 enum TrafficLight sem = GREEN;  
4 int (*fnPtr)(struct Data, union IdLibro, enum TrafficLight) = function;  
5 fnPtr(currDate, bookC, sem);
```

Migliori dichiarazioni di tipi personalizzati:

typedef

```
1 struct Data {
2     unsigned int day;
3     unsigned int month;
4     int year;
5 };
```

```
1 union IdLibro {
2     long ean;
3     char isbn[15];
4 };
```

```
1 enum TrafficLight {
2     GREEN,
3     YELLOW,
4     RED
5 };
```

```
1 int function(
2     struct Data d,
3     union IdLibro b,
4     enum TrafficLight s
5 );
```

Mantenendo le **stesse dichiarazioni** ed introducendo `typedef` nel corpo del programma stesso:

```
1 typedef struct Data Date;
2 typedef union IdLibro BookId;
3 typedef enum TrafficLight TrafficLight;
4 typedef int (*FunctionPtr)(Date, BookId, TrafficLight);
5
6 Date currDate = {14, 10, 1492};
7 BookId bookC = {9780131101630};
8 TrafficLight sem = GREEN;
9 FunctionPtr fnPtr = function;
10 fnPtr(currDate, bookC, sem);
```

Simili alle dichiarazioni di variabili di questi tipi, ma precedute da `typedef`

Simile alla dichiarazione di una variabile puntatore a funzione, ma preceduta da `typedef`

Identificatori
scelti nella
ridefinizione
vengono
usati come
fossero
veri tipi

Migliori dichiarazioni di tipi personalizzati: typedef

17

Modificando *opportunamente* le **dichiarazioni** con

typedef:

```
1 typedef struct {
2   unsigned int day;
3   unsigned int month;
4   int year;
5 } Date;
```

struct
anonima

```
1 typedef union {
2   long ean;
3   char isbn[15];
4 } BookId;
```

union
anonima

```
1 typedef enum {
2   GREEN,
3   YELLOW,
4   RED
5 } TrafficLight;
```

enum
anonimo

```
1 typedef int (*FunctionPtr)(
2   Date, BookId, TrafficLight
3 );
4 int function(
5   Date d,
6   BookId b,
7   TrafficLight s
8 );
```

Ridefinizione classica

Si può scrivere il seguente programma:

Identificatori scelti
nella ridefinizione
vengono usati come
fossero
veri tipi

```
1 Date currDate = {14, 10, 1492};
2 BookId bookC = {9780131101630};
3 TrafficLight sem = GREEN;
4 FunctionPtr fnPtr = function;
5 fnPtr(currDate, bookC, sem);
```

3. Esercizi

Esercizi

19

1. Scrivere una funzione che, sfruttando funzioni della libreria `string.h`, rimuova **tutte le occorrenze** di una sottostringa fornita dall'utente. Si assume che tutte le stringhe siano terminate da `'\\0'`.
Hint: La funzione non dovrebbe avere valore di ritorno e dovrebbe accettare i seguenti parametri: (a) la stringa originale su cui operare, (b) la sottostringa di cui si desidera la rimozione, (c) un puntatore ad una stringa da fornire in output.
Esempio: (a) "ciao ciao ciao", (b) "o" → (c) "cia cia cia"
2. Scrivere due funzioni che verifichino che gli indirizzi di memoria di puntatori a membri di una `struct` siano diversi, mentre gli indirizzi di memoria di puntatori a membri di una `union` siano uguali.
Hint: la format string per puntatori è "%p"; entrambe le funzioni accettano **due argomenti**: (a) un puntatore a struct oppure un puntatore a union, rispettivamente, e (b) un puntatore a funzione che implementi l'**operazione di confronto** (per struct testa uguaglianza tra indirizzi, per union testa disuguaglianza); gli indirizzi di memoria devono essere **confrontati** tramite le funzioni della libreria `string.h`.

4. Allocazione dinamica della memoria

Allocazione dinamica della memoria

21

Utilizzo della memoria nei nostri programmi finora:

- Variabili semplici
- Array statici

```
1 int size = 10;  
2 int array[size];
```

La dimensione di `array` deve essere nota al **momento della dichiarazione**, e dopodiché **non può variare**

- Operazioni su array statici pre-allocati

```
1 int* function(int size){  
2   int array[size];  
3   return array;  
4 }
```

`array` è una **variabile locale** alla funzione, l'allocazione di memoria è valida finché viene usata all'interno della stessa funzione; restituirne il puntatore può sfociare in **undefined behavior**

Allocazione dinamica della memoria

22

La memoria viene suddivisa in due categorie:

➤ Stack

- Statica, tutti gli elementi hanno una **dimensione fissa nota**
- Allocata durante la **compilazione**: variabili, array statici, ... → liberata al **termine** del programma
- Allocata per **chiamate a funzioni**: copie dei parametri, variabili locali, ... → liberata al **return dalla funzione**

➤ Heap

- Dinamica, gli elementi possono avere dimensione variabile nel corso dell'esecuzione del programma
- Allocata a runtime dal programma → liberata **manualmente** dal programma

Allocazione dinamica della memoria (libreria `stdlib.h`)

23

Il linguaggio C fornisce una libreria per la manipolazione dinamica della memoria: `stdlib.h`. Per usare le funzioni disponibili, bisogna importare la libreria tramite l'apposita direttiva al preprocessore `#include`.

Alcune funzioni incluse:

- `malloc(size_t size)`
Alloca una porzione di memoria heap di dimensione `size` bytes
Restituisce un puntatore di tipo `void` (quindi `void*`), indicando un blocco di memoria generico
Il puntatore restituito può essere `NULL` → va controllato prima di usarlo!
- `calloc(size_t num, size_t size)`
Alloca una porzione di memoria heap di dimensione `size` bytes per ognuno dei `num` elementi, inizializzando a `zero` il contenuto del blocco di memoria
Come `malloc`, il puntatore restituito è generico e può essere `NULL`

```
1 int *array = calloc(5, sizeof(int));
```

`calloc` differisce negli argomenti, ma il resto dell'uso è lo stesso

```
1 int *array = malloc(5 * sizeof(int));
2 if(array == NULL){
3     printf("Allocazione fallita\n");
4 }
5 else{
6     for(int i=0; i<5; i++){
7         printf("%d\n", *array);
8         array++;
9     }
10 }
```

`array` è una variabile puntatore ad `int`, che rappresenta il puntatore al primo elemento di un **array di 5 elementi `int`**; il resto delle interazioni rimane identico ad un qualunque puntatore

Allocazione dinamica della memoria (libreria `stdlib.h`)

24

➤ `realloc(void *ptr, size_t new_size)`

Riutilizza la memoria allocata per il puntatore `ptr`, aumentando o diminuendo la dimensione allocata

I contenuti in memoria rimangono intatti

Come `malloc` e `calloc`, viene restituito un puntatore generico se l'allocazione va a buon fine; se tuttavia non è possibile,

il risultato è `NULL` pointer e la precedente porzione di memoria puntata non viene liberata

➤ `free(void *ptr)`

Libera la memoria allocata tramite `malloc`, `calloc`, e `realloc`

Per ogni allocazione è **necessario** utilizzare una chiamata a `free`, in modo da evitare **memory leak** → memoria allocata dal programma e non restituita al sistema operativo

Non deve essere usata su puntatori `NULL` → il **"double free"** corrompe la gestione della memoria (undefined behavior)

```
1 int *array = malloc(5 * sizeof(int));
```



```
1 int *array = realloc(NULL, 5 * sizeof(int));
```


Allocazione dinamica della memoria (libreria `stdlib.h`)

25

L'utilizzo dei meccanismi di allocazione dinamica della memoria permettono quindi l'implementazione di **funzioni che restituiscano nuovi puntatori**, senza incorrere nei problemi di variabili locali deallocate all'uscita dalla funzione.

```
1 int* function(int size){
2     int *array = malloc(size * sizeof(int));
3     return array;
4 }
```

Va però sempre ricordata l'invocazione della funzione `free` sul puntatore ritornato:

```
1 int size = 5;
2 int *array = function(size);
3 if(array == NULL){
4     printf("Allocazione fallita\n");
5     return 1;
6 }
7
8 // Utilizzo di array
9
10 free(array);
```

`array` è un puntatore a
locazione di memoria allocata
dinamicamente nel corpo di
function

Se `array` è `NULL`,
questa funzione non
prosegue nell'esecuzione,
garantendo un'unica
invocazione di `free`

A cosa fare attenzione con la memoria dinamica

26

Errori comuni

- Non controllare **valori di ritorno** delle funzioni di allocazione
- Argomento **size pari a 0** in chiamate a `malloc`
 - Puntatore ritornato potrebbe essere NULL, oppure diverso da NULL con nessuna locazione di memoria associata
 - Se diverso da NULL, non deve essere dereferenziato ma può essere argomento di `free`
- `free` invocata **più volte** sullo stesso puntatore
- `free` invocata su un puntatore **non allocato** tramite `malloc`, `calloc`, o `realloc`
- Dereferenziare un puntatore dopo aver invocato `free` su di esso
- Dimenticare di invocare `free`

Bonus

- Includendo `stdlib.h` non c'è bisogno di usare cast espliciti per i puntatori generici: verranno **automaticamente convertiti** al tipo destinatario

Strutture dati non contigue in memoria

27

Finora:

- Array statici
- `struct`
- Array dinamici

sono tutti allocati in modo **contiguo** in memoria.



Vantaggi

- Accesso agli elementi in tempo costante in qualunque posizione
- Facile gestione del blocco di memoria allocato (es. una sola deallocazione necessaria)
- Generalmente necessita di meno overhead durante il suo utilizzo (es. traduzione indirizzi)

Svantaggi

- Un unico blocco di memoria di grandi dimensioni potrebbe non essere disponibile, causando errori di allocazione
- Frammentazione interna: l'intero blocco di memoria potrebbe non essere utilizzato, sprecando memoria

Strutture dati non contigue in memoria

28

Non tutti i task sono adatti a strutture contigue in memoria:

- Lista a dimensione variabile con **frequenti** inserimenti e/o eliminazioni
- Alberi



Vantaggi

- Ogni elemento occupa una locazione di memoria con dimensione esigua: allocazioni difficilmente falliscono
- Nessuna frammentazione interna: il blocco di memoria allocato è utilizzato totalmente, efficientemente utilizzando la memoria

Svantaggi

- Nessuna garanzia di accesso efficiente in qualunque posizione (dipende dalla struttura dati del caso)
- Gestione memoria non banale
- Possibile presenza di overhead da parte del sistema operativo

Esempio: linked list

29

Linked list:

- Struttura dati non contigua in memoria
- Lista di elementi di dimensione variabile
- Ogni elemento contiene un determinato valore ed il puntatore all'elemento successivo (**single linked list**); può inoltre contenere il puntatore all'elemento precedente (**doubly linked list**)
- Adatta a task in cui la lista degli elementi è usata con frequenti inserimenti e/o eliminazioni:
 - Single linked list → più adatta se inserimenti/eliminazioni sono solo all'inizio o alla fine della lista
 - Doubly linked list → più adatta se inserimenti/eliminazioni possono avvenire in qualsiasi posizione della lista

Esempio: single linked list

30

Dichiarazione elemento base

```
1 typedef struct Element {
2     int value;
3     struct Element *next;
4 } Element;
```

Dichiarazione struct di un **singolo elemento** della lista (in questo caso int)

Inserimento elementi

```
1 head->next = malloc(sizeof(Element));
2 head->next->value = 5;
3 head->next->next = NULL;
```

La lista può essere **estesa** aggiungendo elementi tramite il **puntatore al successivo** elemento

Dichiarazione di una lista

```
1 Element *head = malloc(sizeof(Element));
2 head->value = 0;
3 head->next = NULL;
```

La lista è rappresentata dal **primo elemento**, la *"testa"*

Eliminazione elementi

```
1 Element *ptr = head->next;
2 head->next = ptr->next;
3 free(ptr);
```

Semplicemente viene invocata **free** sul **puntatore all'elemento da eliminare**; è necessario però **aggiornare** il puntatore a **successivo dell'elemento precedente**

5. Esercizi

Esercizi

32

3. Assumendo una **single linked list** composta da `Element` (dichiarato come negli esempi), scrivere le funzioni

`insertAt` e `removeAt` tali che:

- `insertAt` → inserisce un elemento in una qualsiasi posizione, fornita dall'utente come indice (analogo ad array)
- `removeAt` → rimuove un elemento in una qualsiasi posizione, fornita dall'utente come indice (analogo ad array)

*Hint: 1 (head) -> 2 -> 3 -> 4; `insertAt(head, 5, 1)` → 1 (head) -> 5 -> 2 -> 3 -> 4; `removeAt(head, 0)` → 5 (head) -> 2 -> 3 -> 4; prestare attenzione ai **valori di ritorno** delle funzioni e come usarli*

4. Implementare una **doubly linked list** ed annesse **funzioni** tali che:

- Supporti elementi `int`, `float`, e `char[10]` – una lista comunque conterrà elementi di tipi **omogenei**
- Supporti le funzioni `insertAt` e `removeAt` definite come all'Esercizio 3 ed **opportunamente modificate** per l'uso in una `doubly linked list`
- Implementi una funzione `apply` tale che accetti un **puntatore a funzione** `fnPtr` ed una **direzione** (`enum Dir{ FRONT, BACK};`): se `FRONT`, `fnPtr` viene applicato a tutti gli elementi scorrendo la lista **dall'inizio alla fine**, e in **ordine inverso** se la direzione è invece `BACK`

*Hint: è necessario effettuare controlli specifici per evitare elementi di tipi diversi all'interno di una lista; il puntatore a funzione fornito ad `apply` deve aver accesso **solo** ad un **singolo elemento**, potenzialmente con la possibilità di **modificarlo**; può essere utile definire una nuova `struct` che rappresenti la lista stessa, evitando la gestione diretta di*



6. Docker

Sviluppo programmi per sistemi terzi

34

Scrivere e compilare programmi può risultare difficile senza avere accesso ad un sistema simile a quello su cui verranno eseguiti:

- Compilatori specifici
es. Microsoft Visual C++ (msvc), GNU Compiler Collection (gcc), Intel C++ Compiler (icc), distcc, ...
- Differenze di librerie
es. Accesso ai file, multi-processing, multi-threading, ...

Sviluppare direttamente sul sistema di interesse?

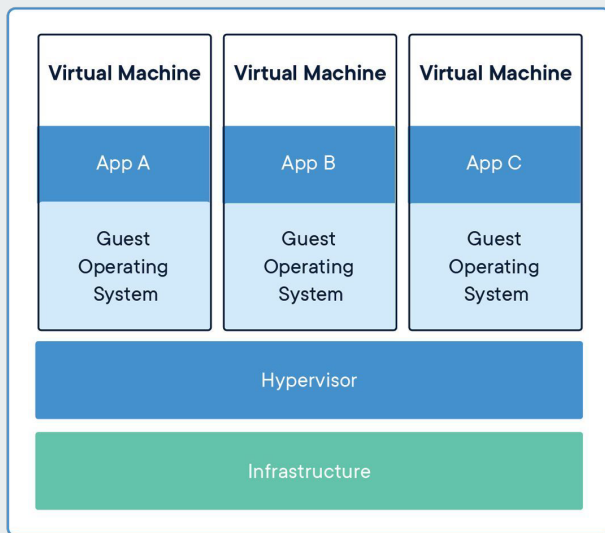
- Stesso sistema operativo
- Architettura hardware simile

Virtualizzazione

35

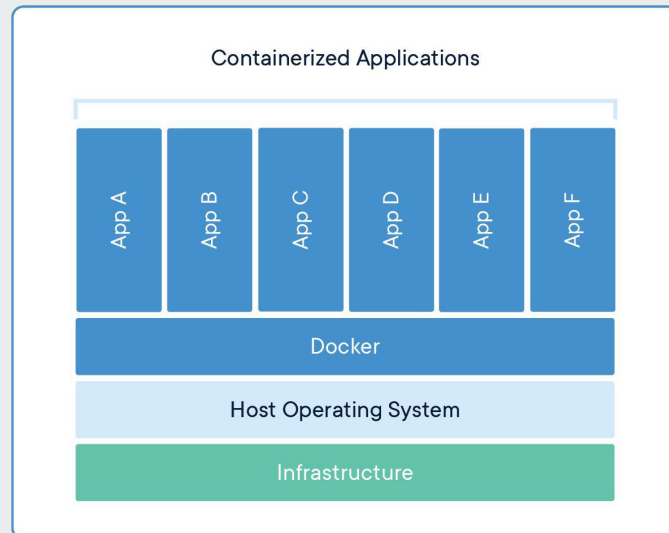
Virtual Machine (VM)

Un macchina virtuale è un metodo per virtualizzare l'intera architettura di un'applicazione, dall'hardware al sistema operativo, con le dovute dipendenze e configurazioni necessarie.



Container

Un container software è un metodo di fornire applicazioni unitamente a tutte le dipendenze necessarie, solamente virtualizzando il sistema operativo.



Docker container

36

Motivazioni:

- Focus su API POSIX nelle prossime lezioni ————— **Necessario** solo per chi usa Windows
- Facile e leggero da eseguire
- Evita l'installazione di un sistema operativo diverso

Installazione Docker:

<https://docs.docker.com/get-docker/>

Docker container

37

Comandi utili dopo aver installato Docker:

- Scaricare immagine Ubuntu 22.04 LTS

```
docker pull ubuntu:22.04
```

```
emanuele@emanuele-Ubuntu:~$ docker pull ubuntu:22.04
22.04: Pulling from library/ubuntu
2ab09b027e7f: Pull complete
Digest: sha256:67211c14fa74f070d27cc59d69a7fa9aeff8e2d
Status: Downloaded newer image for ubuntu:22.04
docker.io/library/ubuntu:22.04
```

- Ottenere l'ID di un'immagine

```
docker images
```

```
emanuele@emanuele-Ubuntu:~$ docker images
REPOSITORY          TAG         IMAGE ID      CREATED       SIZE
ubuntu              22.04      08d22c0ceb15  12 days ago  77.8MB
hello-world         latest     fce289e99eb9  4 years ago  1.84kB
```

- Avviare un container con una data immagine

```
docker run -td --name <nomeCntr> <imgId>
```

```
emanuele@emanuele-Ubuntu:~$ docker run -td --name cont 08d22c0ceb15
aa2b5a9e6084d08f5d8999e0876ba4a131c814ba5ebc93f22dbafc93d88f90
```

- Controllare un container direttamente

```
docker exec -it <cntrId> /bin/bash
```

```
emanuele@emanuele-Ubuntu:~$ docker exec -it aa2b5a9e6084 /bin/bash
root@aa2b5a9e6084:/#
```

- Copiare file all'interno del container (es. sorgenti programmi)

```
docker cp <path1> <cntrId>:<path2>
```

```
emanuele@emanuele-Ubuntu:~$ docker cp source.c aa2b5a9e6084:/home/
Successfully copied 2.56kB to aa2b5a9e6084:/home/
```

Riferimenti

38

- <https://en.cppreference.com/w/c/string/byte>
- <https://en.cppreference.com/w/c/language/struct>
- <https://en.cppreference.com/w/c/language/union>
- <https://stackoverflow.com/questions/252552/why-do-we-need-c-unions>
- <https://en.cppreference.com/w/c/language/enum>
- <https://en.cppreference.com/w/c/language/typedef>
- <https://en.cppreference.com/w/c/memory>
- [https://en.wikipedia.org/wiki/Containerization \(computing\)](https://en.wikipedia.org/wiki/Containerization_(computing))
- <https://www.docker.com/resources/what-container/>
- <https://docs.docker.com/get-docker/>
- https://hub.docker.com/_/ubuntu/tags