TIPI STRUTTURATI: STRUCT E ENUM

Roberto Nardone, Luigi Romano

ROADMAP

- Enumerazioni
- Definizioni di tipo con typedef
- Perché e in che casi utilizzare le Strutture
- Caratteristiche peculiari delle Strutture
- Accesso alle strutture
- Funzioni membro delle strutture
- Puntatori a strutture

ENUMERAZIONI: ENUM – 1/2

- Nei problemi reali può succedere di dover trattare *colori*, *categorie*, *dimensioni*, ecc., per i quali non esiste un tipo predefinito che si adatti opportunamente alla loro rappresentazione.
- In questi casi il C++ permette di definire le entità su cui il programma deve operare:
  - Questa possibilità è data dalla definizione dei **tipi enumerativi**.
- Ogni enumerazione è un **tipo**.
- I valori che possono essere assunti da una variabile enumerazione sono ristretti ad un insieme di valori **mappati su interi costanti**.

ENUMERAZIONI: ENUM – 2/2

- Il formato di un **enum** è il seguente:

```
enum <identificativo> {
        <enumeratore>,
        … ,
        <enumeratore>
        };
```

- Ad esempio:

```
enum  Colore  {bianco, rosso, blu, giallo, verde, nero};
enum GiornoSettimana {lunedi, martedi, …, domenica};

……..
Colore  col_parete;
GiornoSettimana giorno;
```

- La variabile **col_parete**, di tipo **Colore**, potrà assumere i valori **bianco, rosso, blu**, ecc.
- La variabile **giorno** di tipo **GiornoSettimana** i valori **lunedi, martedi**, ecc.

CODIFICA VALORI DI UN ENUM – 1/3

- In un programma si potranno avere, dunque, istruzioni del tipo:

```
col_parete = verde;
giorno = mercoledi;
```

- Nella definizione di tipo si evidenziano due elementi: l' **identificatore di tipo** (colore), detto **tag**, e gli identificatori che verranno usati per esprimere i **valori assunti** (lunedì) da quel tipo.
- I valori assunti da quel tipo corrispondono a delle **costanti intere definite in modo automatico dal compilatore**, che pertanto devono essere uniche nel programma. Ad esempio:

```
enum Festivo{sabato,domenica};
enum Feriale{lunedi, martedi, ..., sabato};
```

- E' **errato** perché **sabato** è presente in due tipi; analogamente è illecito usare, per esempio, un enumeratore come nome di una variabile.

CODIFICA VALORI DI UN ENUM – 2/3

- La lista degli identificatori che denotano i valori assunti da un tipo enumerato forma un insieme ordinato che il compilatore **codifica** con **valori interi crescenti a partire da 0**.

```
enum  Colore  {bianco, rosso, blu, giallo, verde, nero};
```

- Nella definizione del tipo **Colore**, l'identificatore **bianco** viene codificato con **0**, **rosso** con 1, **blu** con 2, ecc.
- Si possono modificare questi valori, assegnati automaticamente, imponendo i valori interi che si vogliono assegnare direttamente nella definizione.
- Si fa seguire al nome dell'identificatore il segno uguale e il valore che si intende assegnare. Il compilatore assumerà questo numero come nuovo valore di partenza per le assegnazioni successive.

CODIFICA VALORI DI UN ENUM – 3/3

- Ad esempio:

```
enum  Colore  {bianco, rosso, blu=20, giallo, verde, nero};
```

- In questo caso bianco vale 0, rosso vale 1, blu vale 20, giallo vale 21, ecc.

- E' possibile attribuire a più identificatori valori identici, o esplicitamente (ad esempio, blu = 20, giallo = 20), o implicitamente: se, ad esempio, si pone blu = 0 (lo stesso valore di bianco che a questo punto è già stato codificato), giallo assumerà lo stesso valore di rosso, cioé 1.

ENUMDEMO.CPP


```cpp
//  EnumDemo.cpp
//  Description:
//    Demonstrates enum type.
//  Author: lrom - 23 Sep, 2001

// For older compilers
// #include <iostream.h>

// For recent compilers
#include <iostream>
using namespace std;

int main (void)
{
  typedef enum {LUN, MAR, MER, GIO, VEN, SAB, DOM}Giorno;

  Giorno g = LUN;
  cout << endl << "g = "   << g   << endl;
  cout << endl << "g+1 = " << g+1 << endl;

  g=MAR;
  cout << endl << "g = "   << g   << endl;
  cout << endl << "g+1 = " << g+1 << endl;

  g=1;

  return 0;
}
```

```
ubuntu@ubuntu:~/cppprogs$ g++ -o EnumDemo.exe EnumDemo.cpp
EnumDemo.cpp: In function ‘int main()’:
EnumDemo.cpp:25:5: error: invalid conversion from ‘int’ to ‘main()
::Giorno’ [-fpermissive]
   g=1;
     ^
```

ENUM DIETRO LE QUINTE


```
enum  Colore  {bianco=5, rosso, blu=20, giallo, verde, nero} c;
```

- In realtà Colore verrà definito come un intero che può assumere valori tra la più grande potenza di due <= del minimo valore attribuito e la più piccola potenza di due >= del massimo valore attribuito
  - Colore assume valori compresi tra $5>2^2$ e $23<2^5$ pertanto, in realtà, il range di valori che può assumere colore è compreso tra 2^2 e 2^5 quindi, ad esempio `c = 18` non darà errore ma sarà corretta

ESEMPIO USO ENUM


```
6  enum Giorno
7  {
8      lun = 1,
9      mar,
10     mer,
11     gio,
12     ven,
13     sab,
14     dom
15 };
16 void stampa(Giorno g){
17     switch( g )
18     {
19         case lun: case mar: case mer: case gio: case ven:
20             cout << "lavorativo"<<endl;
21             break;
22
23         case sab: case dom:
24             cout << "feriale"<<endl;
25             break;
26     }
27 }
28 int main()
29 {
30     Giorno g;
31
32     g = lun;
33     stampa(g);
34     g = dom;
35     stampa(g);
36
37     return 0;
38 }
39
```

DEFINIZIONI DI TIPO: TYPEDEF

- E' possibile ridefinire un tipo già esistente mediante la parola chiave **typedef**, cosicché si può utilizzare un nuovo identificatore (sinonimo) al posto del precedente.
- Formato:

  typedef *<tipo_esistente>* *<nuovo_nome>*;

- Esempio:

```
typedef long int interolungo; //ridefinizione del tipo
interolungo dato_int; //dato_int è un intero lungo
```

ESEMPIO USO TYPEDEF

- Come già detto nella lezione sui tipi di dato, ogni tipo richiede una certa quantità di memoria (in bit) per rappresentare il dato.
- La rappresentazione in bit varia a seconda del calcolatore su cui si compila il programma
- Mediante typedef si possono definire, ad esempio, tipi astratti, integer8, integer16 e integer32, dall'ovvio significato, per poi ridefinirli in funzione del compilatore effettivamente usati.

```
typedef long long int int_64;
typedef short int int_16;
typedef char C;

int_64 var=0;
C carattere='a';
```

PERCHÈ LE STRUTTURE? – 1/3

- Supponiamo di voler memorizzare in una variabile i dati di alcune persone.
- Per ogni persona, abbiamo bisogno di una stringa per il nome, una stringa per il cognome e un intero per l'età.
- Un semplice programma basato su queste variabili è il seguente…

PERCHÈ LE STRUTTURE? – 2/3


```
#include <iostream>

using namespace std;

void stampaPersona(char*, char*, int);

int main() {

        char nome[] = "Francesco";
        char cognome[] = "Verdi";
        int eta = 43;

        stampaPersona(nome, cognome, eta);

        return 0;
}

void stampaPersona(char* nome, char* cognome, int eta) {
        cout<<nome<<" "<<cognome<<", "<<eta;
}
```

- Il nome e il cognome della persona vengono memorizzati tramite stringhe C-style, chiamate nome e cognome, rispettivamente. L'età viene invece memorizzata nella variabile intera eta.

PERCHÈ LE STRUTTURE? – 3/3

- Tuttavia, il programma presenta un problema di:
  - **Leggibilità,** non risulta subito chiaro che le tre variabili nome, cognome, ed eta sono tre attributi dello stesso oggetto fisico.
  - **Modificabilità**, se vogliamo aggiungere la data di nascita alle informazioni da memorizzare, bisogna aggiungere una variabile, e poi modificare tutte le chiamate di funzione.
- Serve uno strumento che consenta di **definire una variabile come un gruppo di variabili anche di tipo diverso**

LE STRUTTURE – 1/3

- Una **struttura dati** è un insieme di tipi diversi di dati, raggruppati in un'unica dichiarazione

```
struct nome_modello {
        tipo1 nome_elemento1;
        tipo2 nome_elemento2;
        tipo3 nome_elemento3;
 } nome_oggetto;
```

- Il *nome_modello* è un nome per il modello di struttura e *nome_oggetto* (opzionale) è un identificatore che denota un oggetto avente la struttura *nome_modello*
- Tra le parentesi graffe **{}** sono indicati i **campi** della struttura
- In particolare, i tipi e i rispettivi sub_identificatori degli elementi che compongono la struttura.

LE STRUTTURE – 2/3

- Una volta dichiarata una struttura, il suo nome può essere **utilizzato come un nuovo tipo di dati**. Ad esempio:

```
struct Persona {
        char nome[30];
        char cognome[30];
        int eta;
};
.............
Persona p;
```

- Abbiamo definito il modello di struttura **Persona** con tre **campi**: **nome, cognome** e **eta**, di tipo diverso.
- Abbiamo quindi usato il nome del modello di struttura (**Persona**) per dichiarare la variabile p di tale tipo.

LE STRUTTURE – 3/3

- Una volta dichiarato, **Persona** è diventato un nuovo nome di tipo al pari di quelli fondamentali come **int** , **char** o **short**.
- Abbiamo quindi potuto successivamente dichiarare degli **oggetti** (variabili) di tale tipo.
- Un altro modo di dichiarare la struttura precedente, utilizzando il *nome_oggetto*, è il seguente:

```
struct Persona {
        char nome[30];
        char cognome[30];
        int eta;
} p;
```

- Il campo opzionale *nome_oggetto* che compare alla fine della dichiarazione di una struttura serve a dichiarare direttamente oggetti di tale tipo.

MODELLO ED OGGETTO

- In questo caso, in cui inseriamo direttamente nella dichiarazione della struttura la dichiarazione di tutti gli oggetti di tale tipo, il nome *nome_modello* (**Persona** nel nostro caso) è opzionale.
- Distinguere chiaramente tra i concetti di ***modello*** della struttura e di ***oggetto*** appartenente alla struttura.
- Facendo il parallelo con i termini che abbiamo usato per le variabili possiamo dire che il **modello** (nel nostro caso Persona) è il **tipo** e l'oggetto (nel nostro caso p) è la variabile. Si possono dichiarare molti oggetti (variabili) di uno stesso modello (tipo).
  - **Modello ←→ Tipo**
  - Oggetto ←→ Variabile

ACCESSO AI CAMPI DELLE STRUTTURE

- Una volta dichiarati gli oggetti (nell'esempio **p**) possiamo operare sui **campi** che li costituiscono.
- Per fare questo bisogna usare un punto (.) tra il nome dell'oggetto ed il nome del campo.
- Ad esempio possiamo operare con le seguenti notazioni esattamente come fossero degli identificatori di variabile appartenenti ai rispettivi tipi:

  p.nome
  p.cognome
  p.eta

- Ciascuna di esse appartiene al rispettivo tipo: **p.nome** e **p.cognome** sono di tipo **char[30]**, mentre **p.eta** è di tipo **int**.

ESEMPIO USO STRUTTURE – 1/2


```
#include <iostream>

using namespace std;

struct Persona {
        char nome[30];
        char cognome[30];
        int eta;
};

void stampaPersona(Persona p);

int main() {
        Persona pers = {"Francesco", "Verdi", 43};



        stampaPersona(pers);

        return 0;
}

void stampaPersona(Persona p) {
        cout<<p.nome<<" "<<p.cognome<<", "<<p.eta;
}
```

```
#include <iostream>

using namespace std;



void stampaPersona(char*, char*, int);

int main() {

        char nome[] = "Francesco";
        char cognome[] = "Verdi";
        int eta = 43;

        stampaPersona(nome, cognome, eta);

        return 0;
}

void stampaPersona(char* nome, char* cognome, int eta) {
        cout<<nome<<" "<<cognome<<", "<<eta;
}
```

PUNTATORI A STRUTTURE

- Un puntatore può anche puntare una struttura. Ad esempio:

```
struct Persona{
   string nome;
   string cognome;
   int altezza;
   int peso;
};

void stampaPersona(Persona p);

int main(){
   Persona persona = {"Pippo","Pluto",180,80};
   Persona *pp;

   pp = &persona;
```

ACCESSO AI CAMPI DELLE STRUTTURE TRAMITE PUNTATORI

- Come visto nella scorsa lezione, per accedere ad un campo della struttura, si deve indicare il nome della variabile seguito dal punto e dal campo di interesse. Ad es:
  - **persona.nome**
- Se vogliamo utilizzare il puntatore alla struttura, allora dovremo dereferenziarlo. E dunque:
  - **(*pp).nome**
- Da notare che usiamo le parentesi poichè l'operatore punto ha precedenza sull'asterisco
- Siccome, tale metodologia risulta alquanto scomoda, il C++ permette l'uso di un operatore di freccia (->) a sostituzione delle parentesi tonde, del punto e dell'asterisco.
  - **pp->nome**

ESERCIZI

- Immagazzinare le seguenti informazioni su una persona: nome, età, altezza, peso. Scrivere una funzione che legga i dati di una persona, ricevendo come parametro un puntatore e scrivere un'altra funzione che li visualizzi.
- Si scriva un programma che definisce un'automobile, caratterizzata da marca, targa, numero cavalli, e tipologia (*nuova* o *usata*).
  - Il programma dovrà includere una funzione di inserimento dati dell'auto ed una funzione di stampa auto *nuove* e stampa auto *usate*.
  - Linee guida:
    - Utilizzare una **struct** *auto* per rappresentare l'automobile
    - Utilizzare un **enum** per rappresentare la tipologia
    - Utilizzare un **vettore** di **struct** *auto* per raccogliere tutte le auto inserite dall'utente

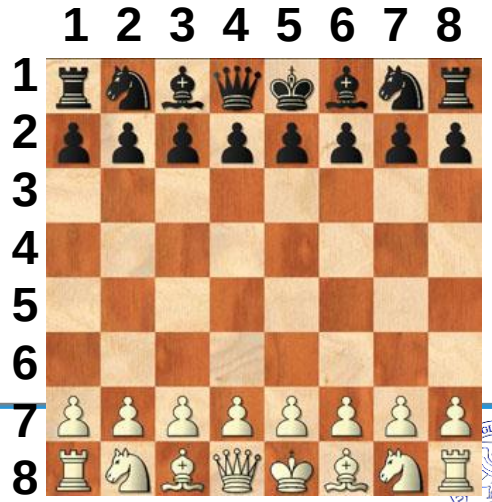
ESERCIZIO AVANZATO

- Scrivere un programma che consenta di inserire e stampare il layout di una scacchiera durante una partita di scacchi. L'inserimento dei pezzi avverrà specificando la quadrupla: tipo pezzo colore riga colonna. Possono essere inseriti al più 32 pezzi, per terminare l'inserimento si può usare il carattere '#'.
  Tipo Pezzo: può assumere uno dei seguenti valori P = Pedone A = Alfiere T = Torre C = Cavallo K = Re R = Regina; il colore che può essere B = Bianco e N = Nero. Riga e Colonna assumono (1, 1) come casella in alto a sinistra. La funzione di inserimento del layout deve dare errore se: ci sono più di 8 pedoni dello stesso colore, se ci sono più di due Alfieri/Cavalli/Torri di uno stesso colore, se ci sono più di un Re o una regina di un colore (Errore 1). Assumendo che il bianco parta dal basso e il nero dall'alto, è errore se un pedone bianco si trova sulla riga 8 o uno nero sulla riga 1 (Errore 2). E' errore se non c'è uno dei due RE (Errore 3). E' errore se una casella è occupata da due pezzi (Errore 4)
  Definire una funzione per stampare la scacchiera usando le minuscole per i bianchi e le maiuscole per i neri (usare un carattere seguito da uno spazio per ciascun pezzo). Due caratteri ' ' (spazio vuoto) per le caselle nere due caratteri '▒' (ASCII 219) per le caselle bianche.

- Esempio:

  P B 2 2 (Pedone Bianco in 2 2)

  A N 4 2 (Alfiere Nero in 4 2)

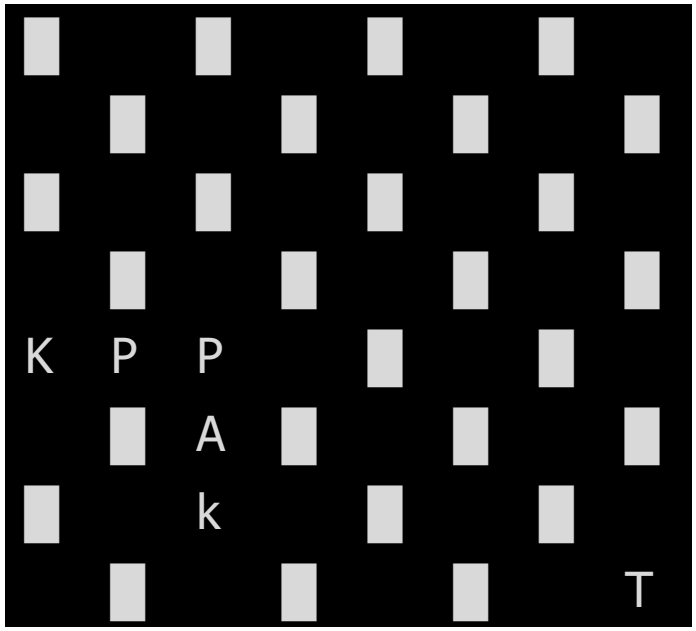


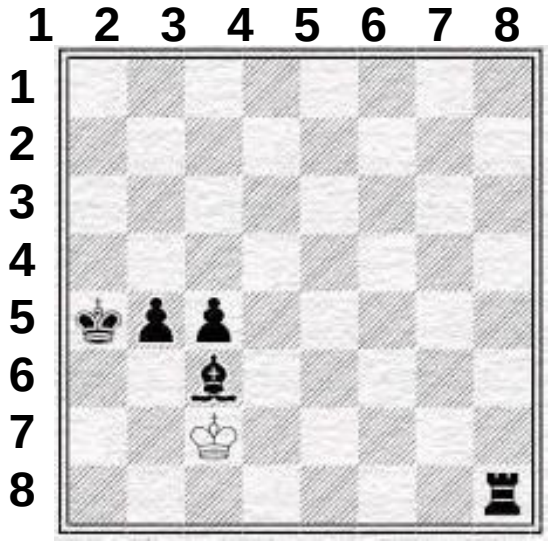
CASI DI TEST

- Caso 1:

```
K N 5 1
P N 5 2
P N 5 3
A N 6 3
K B 7 3
T N 8 8
#
```

Output 1:

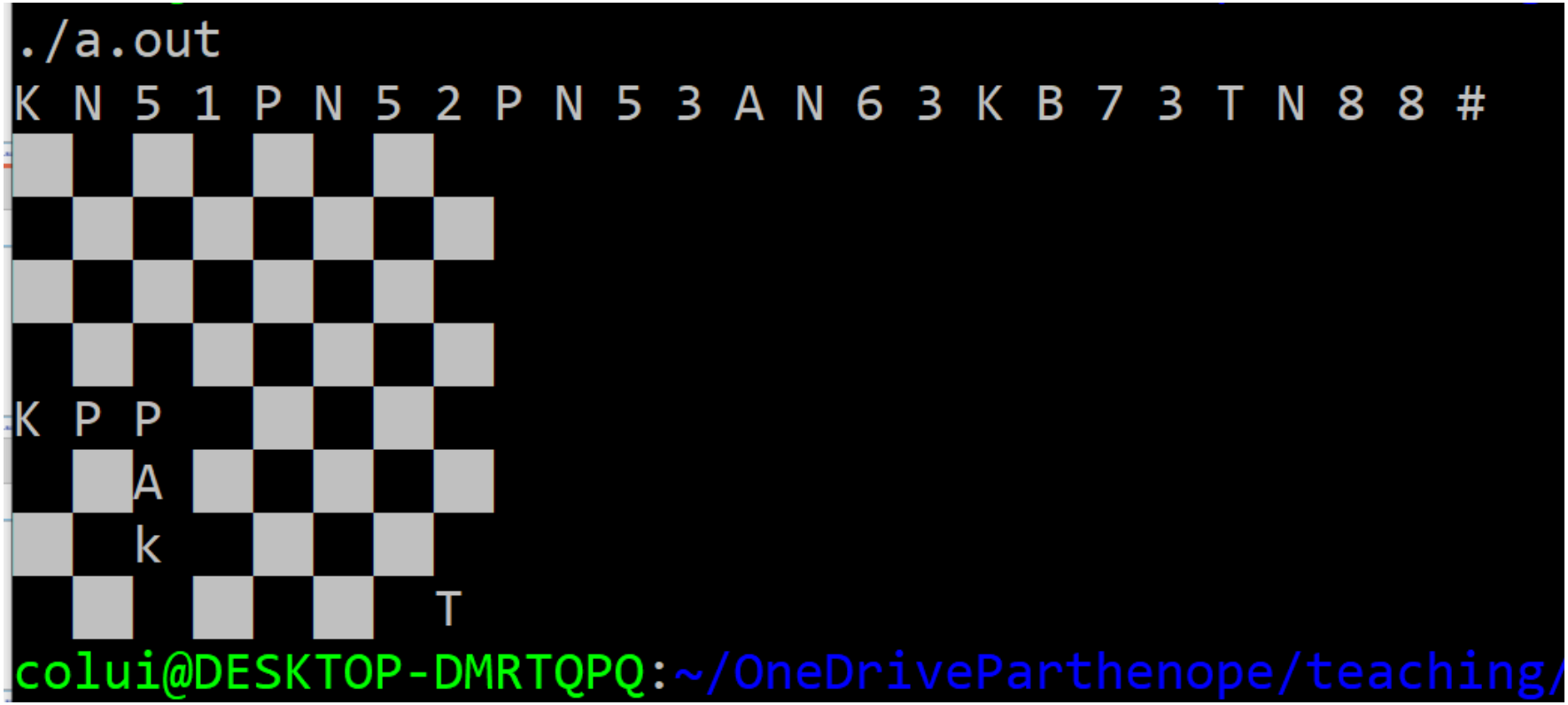




= ASCII 219

Sarà bianco su fondo scuro e scuro su fondo bianco

ESECUZIONE CASO 1



VERIFICARE I CASI DI TEST

- ./a.out < ./test1.in > prova.out
- diff prova.out test1.out

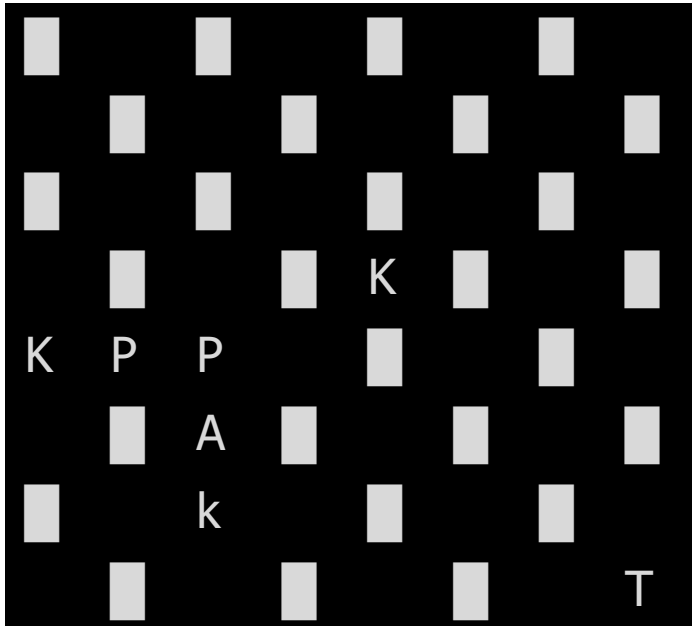
- Se l'uscita è vuota il test è passato

CASI DI TEST

- Caso 2:

```
K N 5 1
P N 5 2
P N 5 3
A N 6 3
K B 7 3
K N 4 5
T N 8 8
#
```

Output 1:
Errore 1



CASI DI TEST

- Caso 3:

  K N 5 1
  P N 5 2
  P N 5 3
  A N 6 3
  T N 8 8
  #

Output 1:
Errore 3