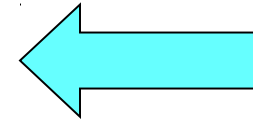


Lezione 15

Stringhe
Struct
Matrici statiche

Contenuto lezione

- Stringhe
- Struct
 - Operazioni
 - Progettazione strutture dati e passaggio parametri
- Matrici statiche
 - Implementazione
 - Passaggio alle funzioni



Stringa 1/2

- Una *stringa* è una sequenza di caratteri
 - Questa è la definizione di un oggetto (tipo di dato) astratto
 - Vedremo a breve come implementarlo con oggetti concreti
- Letterale stringa (costante senza nome): sequenza di caratteri che costituisce la stringa, delimitata da doppi apici
- Esempio: la costante letterale per la stringa *sono una stringa* è **"sono una stringa"**
- All'oggetto *cout* abbiamo spesso passato dei letterali di tipo stringa mediante l'operatore di uscita <<

Stringa 2/2

- Utilizzando le stringhe si possono iniziare finalmente a scrivere programmi dall'output un po' più accattivante
- Ad esempio, uno dei compiti per casa sulle stringhe sarà un programma che chiede all'utente di inserire una parola da *stdin*, e ristampa sul terminale tale parola, trasformando in una lettera maiuscola il primo carattere nel caso sia una lettera minuscola, ed utilizzando i font di default del tool *figlet* (o di tool equivalenti)

Parola da stampare: paolo



paolo

Altri esempi costanti stringa

`"Hello\n"`

`"b"` (stringa contenente un solo carattere, per il momento la consideriamo equivalente a `'b'`)

`""` **stringa nulla**

Stringhe in C/C++

- Nel linguaggio C/C++ non esiste propriamente il tipo stringa
- E' implementato concretamente mediante
 - Un array di caratteri terminati da un carattere **terminatore**
 - Il terminatore è il carattere speciale '`\0`'
 - Numericamente, il suo valore è 0
- Come per i vettori, nella libreria standard del C++ è disponibile anche un tipo astratto stringa (*string*) con interfaccia di più alto livello di un array di caratteri
 - In questo corso non vedremo tale tipo astratto

Sintassi definizione

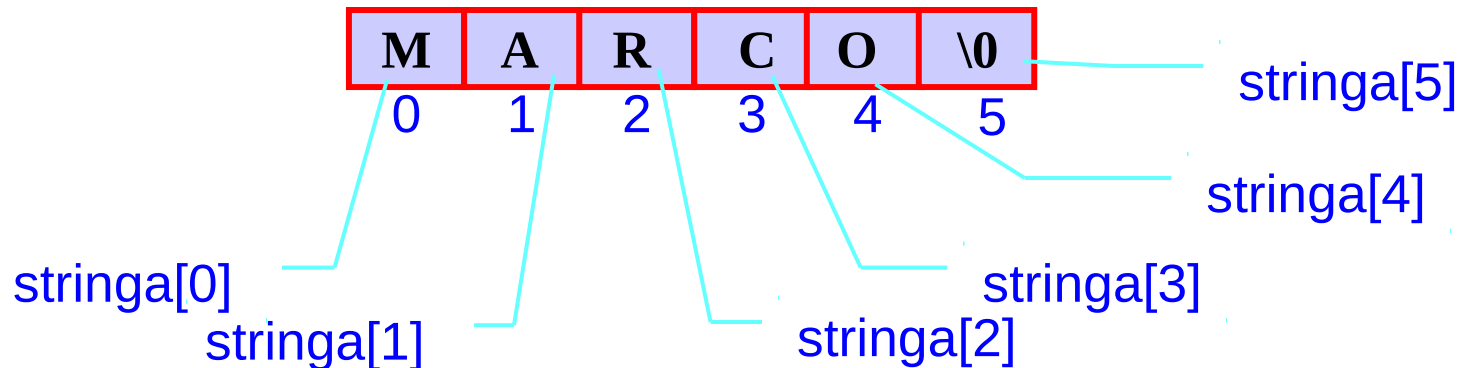
- SINTASSI della definizione di un oggetto di tipo stringa:

```
[const] char <identificatore> [ <espr-costante> ] ;
```

- Esempio:

```
char stringa[6] ;
```

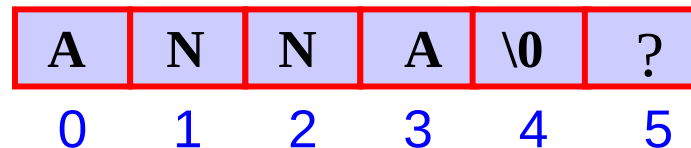
- alloca spazio per 6 oggetti di tipo **char**
- uno va utilizzato per il fine stringa '**\0**'
- quindi la stringa ha al più 5 caratteri



- L'istruzione `char stringa[N] ;`
 - Alloca spazio per una stringa di al più $N-1$ caratteri
 - E' possibile memorizzare in un array di N caratteri anche una stringa di dimensione inferiore ad $N-1$

Esempio:

```
char nome[6] ;
```



- in questo caso le celle oltre il carattere '\0' sono **concettualmente vuote**
 - ovviamente contengono pur sempre un valore, che però non viene preso in considerazione

- Una stringa è implementata mediante un *array* di caratteri
- Ma un *array* di caratteri non è necessariamente l'implementazione di una stringa
- Affinché un *array* di caratteri implementi una stringa, è **necessario** che contenga il terminatore '`\0`'

Inizializzazione

- Vi sono tre modi per inizializzare una stringa in una definizione:

```
char nome[6] = { 'M', 'A', 'R', 'C', 'O', '\0' } ;  
/* come un normale array */
```

```
char nome[6] = "MARCO" ;  
/* sintassi utilizzabile solo per le stringhe; il  
carattere di fine stringa viene inserito  
automaticamente dal compilatore */
```

```
char nome[] = "MARCO" ;  
/* in questo caso la stringa viene dimensionata  
automaticamente a 6 ed il carattere di fine  
stringa viene inserito dal compilatore */
```

Assegnamento

- Se non si tratta di una inizializzazione, l'unico modo per assegnare un valore ad una stringa è carattere per carattere (come un normale array), con esplicito inserimento del carattere di fine stringa:

```
char nome[6];  
nome[0]= 'M';  
nome[1]= 'A';  
nome[2]= 'R';  
nome[3]= 'C';  
nome[4]= '0';  
nome[5]= '\\0';
```

Input/output di stringhe

- Se un oggetto di tipo stringa (ossia array di caratteri)
 - viene passato al *cout/cerr* mediante l'operatore <<
 - vengono stampati tutti i caratteri dell'array, finché non si incontra il terminatore
 - viene utilizzato per memorizzarvi ciò che si legge da *cin* mediante l'operatore >>
 - vi finisce dentro la prossima **parola**, ossia sequenza di caratteri non separati da spazi
Esempio: se sullo *stdin* vi è "ciao mondo", nella stringa finisce solo "ciao" (e sullo *stdin* rimane " mondo")
- Esercizio: definire un oggetto di tipo stringa, inizializzarlo, stamparlo, riversarvi dentro il contenuto dello *stdin*, ristamparlo

Soluzione

```
main()
{
    const int MAX_LUN = 20 ;
    char stringa[MAX_LUN] = "prova";
    cout<<stringa<<endl ;
    cin>>stringa ;
    cout<<stringa<<endl ;
}
```

Domanda

- Che succede se l'utente immette una parola più lunga delle dimensioni massime della stringa che avete definito nel programma?

Domanda

- Utilizzando il manipolatore *noskipws* si riesce a leggere più di una parola alla volta con l'operatore di ingresso?

- Definire un array di caratteri ed inizializzarlo successivamente come una stringa
 - ESEMPIO DI SEQUENZA DI ISTRUZIONI ERRATA:
`char nome[6];`
`nome = "MARCO" ;` **NO**
- Copiare una stringa in un'altra con l'operazione di assegnamento
 - ESEMPIO DI SEQUENZA DI ISTRUZIONI ERRATA:
`char nome[15], cognome[15] ;`
`nome = cognome;` **NO**
- In conclusione, come abbiamo già visto parlando degli array, gli elementi vanno copiati uno alla volta

Domanda

- C'è differenza tra 'A' ed "A" ?
- Occupano lo stesso spazio in memoria?

Caratteri e stringhe

- `'A'` carattere *A*, rappresentabile in un oggetto di tipo `char`, ad esempio
`char c = 'A' ;`
- `"A"` stringa *A*, rappresentabile in un array di due caratteri, ad esempio
`char s[2] = "A" ;`
- Tale differenza ha un impatto anche sulla rappresentazione in memoria:

A

A \0

Stringa ed elementi

- I singoli caratteri di una stringa possono anche essere visti come oggetti indipendenti
"MARCO" → 'M' 'A' 'R' 'C' 'O' '\0'
- Se pensati come stringa sono però parte di un tutt'uno

Stringhe statiche e dinamiche

- Nell'accezione comune, una stringa è una sequenza di caratteri la cui lunghezza può variare
 - Per supportare stringhe dinamiche di qualsiasi lunghezza bisognerebbe utilizzare l'allocazione dinamica della memoria
- Poiché tale argomento sarà trattato in seguito, per ora si prenderà in considerazione solo il caso di
 - stringhe statiche (dimensione fissa)
 - stringhe dinamiche con dimensione massima definita a tempo di scrittura del programma

Domanda

- Cosa stampa il seguente programma?

```
main()
{
    char a[] = {'c', 'i', 'a', 'o' } ;

    for (int i = 0; ; i++)
        cout<<a[i] ;
}
```

Esercizio

- Scrivere un programma che legga una parola da *stdin* e ne stampi la lunghezza
 - Senza utilizzare funzioni di libreria per le stringhe
- Ripetere l'esercizio utilizzando invece una stringa contenente più parole, ed inizializzata a piacere

- Scandire tutto l'array che rappresenta la stringa fino al carattere di terminazione '`\0`', contando i passi che si effettuano

Algoritmo e struttura dati

- Algoritmo
 - Inizializzare una variabile contatore a 0
 - Ripetere un ciclo fino al carattere '`\0`' ed incrementare la variabile contatore ad ogni passo
 - Stampare il valore finale della variabile contatore
- Struttura dati
 - Una variabile per memorizzare la stringa
 - Una variabile ausiliaria come indice del ciclo e forse un'ulteriore variabile come contatore del numero di caratteri ...

Programma

```
main()
{
    int conta=0;
    char dante[]="Nel mezzo del cammin di nostra vita";

    for (int i=0; dante[i]!='\0'; i++)
        conta++; // poteva bastare la sola variabile i

    cout<<"Lunghezza stringa = "<<conta<<endl ;
}
```

Domanda

```
main()
{
    int conta=0;
    char dante[]="Ho preso 0 spaccato";

    for (int i=0; dante[i] != '\0'; i++)
        conta++;

    cout<<"Lunghezza stringa = "<<conta<<endl ;
}
```

E' corretto?

- Sì, perché il codice del carattere `'\0'` è diverso dal codice del carattere `'0'`

Esercizio

- Scrivere un programma che legga una parola da *stdin* e ne assegni il contenuto ad un'altra stringa
 - la stringa di destinazione deve essere memorizzata in un *array* di dimensioni sufficienti a contenere la stringa sorgente
 - il precedente contenuto della stringa di destinazione viene perso (sovrascrittura)
- Ripetere l'esercizio utilizzando invece una stringa contenente più parole, ed inizializzata a piacere

Algoritmo e struttura dati

- Algoritmo
 - Scandire tutta la prima stringa fino al carattere di terminazione '\0'
 - Copiare carattere per carattere nella seconda stringa
 - **Aggiungere il carattere di fine stringa!**
- Struttura dati
 - Due variabili stringa ed almeno un indice per scorrere gli array

Programma

```
main()
{
    int i; // volutamente non definito nell'intestazione del for
    char origine [] = "Nel mezzo del cammin di nostra vita";
    char copia [40] ;

    for (i=0; origine[i] != '\0' ; i++)
        copia[i]=origine[i];    /* si esce prima della copia del
                                carattere di fine stringa che,
                                quindi, va aggiunto
                                esplicitamente */

    copia[i]='\0' ; // FONDAMENTALE !!!
    // da qui in poi si può utilizzare copia come una stringa
    ...
}
```

Stampa di una stringa

- Una stringa si può ovviamente stampare anche carattere per carattere

Esempio:

```
int i=0;
char str[]=
    "Nel mezzo del cammin di nostra vita";
...
while (str[i] != '\0') {
    cout<<str[i];
    i++;
}
```

Passaggio alle funzioni

- Per le stringhe valgono la stessa sintassi e semantica del passaggio alle funzioni degli *array*
 - Sono quindi passate sempre per riferimento
 - E' opportuno utilizzare il qualificatore **const** per un parametro formale di tipo stringa che non viene modificato dalla funzione

Domanda

- Quando si passa una stringa ad una funzione, è sempre necessario passare anche la lunghezza della stringa?

- No, perché c'è il terminatore

Funzioni di libreria

- Così come per le funzioni matematiche e quelle sui caratteri, il linguaggio C/C++ ha una ricca libreria di funzioni per la gestione delle stringhe, presentata in `<cstring>` (`string.h` in C)
- **`strcpy(stringa1, stringa2)`**
copia il contenuto di `stringa2` in `stringa1` (sovrascrive)
- **`strncpy(stringa1, stringa2, n)`**
copia i primi `n` caratteri di `stringa2` in `stringa1`
- **`strcat(stringa1, stringa2)`**
concatena il contenuto di `stringa2` a `stringa1`
- **`strcmp(stringa1, stringa2)`**
confronta `stringa2` con `stringa1`: 0 (uguali), >0 (`stringa1` è maggiore di `stringa2`), <0 (viceversa)

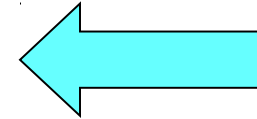
- Svolgere la nona esercitazione fino ai tipi strutturati esclusi

Esercizi senza soluzione

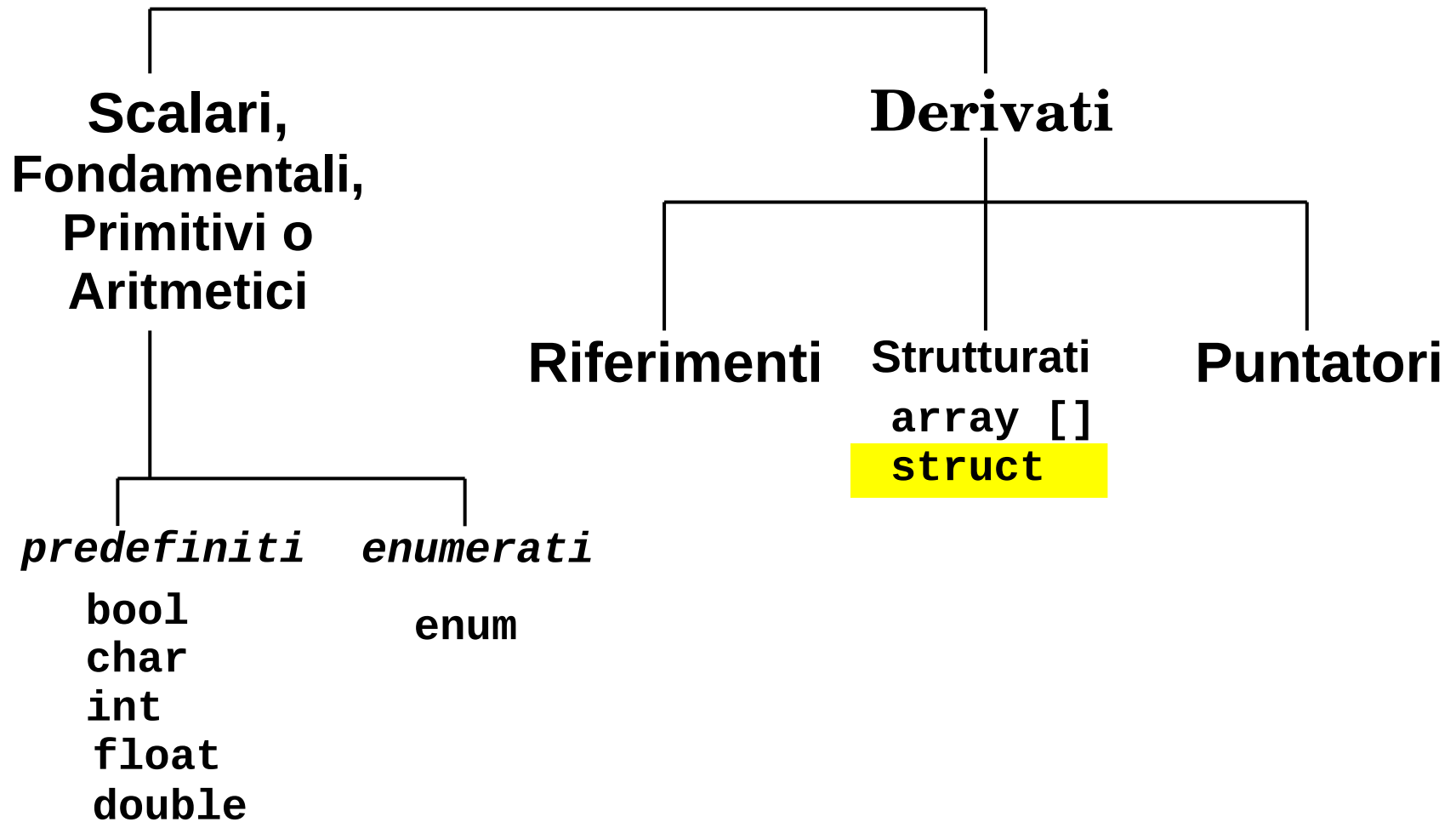
- Controllare se una stringa è più lunga di un'altra
- Copiare soltanto i primi 10 caratteri di una stringa in un'altra stringa, inizialmente vuota. [Attenzione: esistono?]
- Copiare soltanto le vocali di una stringa in un'altra stringa, inizialmente vuota.
- Copiare soltanto le lettere minuscole di una stringa in un'altra stringa, inizialmente vuota.
- Concatenazione (append) di due stringhe: Aggiungere una stringa in fondo ad un'altra stringa, lasciando uno spazio tra le due stringhe [Att.!: la seconda stringa può essere vuota o no]
- Verificare se due stringhe sono uguali o diverse
- Data una frase, contare il numero dei caratteri maiuscoli, minuscoli, numerici e dei caratteri non alfanumerici

Contenuto lezione

- Stringhe
- Struct
- Operazioni
- Progettazione strutture dati e passaggio parametri
- Matrici statiche
 - Implementazione
 - Passaggio alle funzioni



Tipo di dato struttura



Problema 1/2

- Dobbiamo scrivere un programma che lavori su delle 'persone'
- In particolare, per ogni persona, nel programma si devono manipolare i seguenti dati:
 - Nome (stringa di al più 15 caratteri)
 - Cognome (stringa di al più 20 caratteri)
 - Luogo di nascita (stringa di al più 20 caratteri)
 - Età (int)
 - Altezza espressa in metri (double)
 - Codice fiscale (stringa di 16 caratteri)

Problema 2/2

- Come facciamo a memorizzare i dati di più persone?
- Se per esempio il programma avesse lavorato **solo** sull'altezza di varie persone, quale tipo di dato avremmo potuto utilizzare per rappresentare tale informazione per tutte le persone?

- Un *array* di *double*
 - Col quale implementare magari un vettore dinamico con le tecniche che conosciamo
- Purtroppo però ogni persona è caratterizzata da più di un attributo!
- Come potremmo generalizzare la precedente soluzione continuando ad utilizzare solo i tipi di dato che conosciamo?

- Utilizzando un array per ogni attributo, quindi
 - Un *array* di nomi
 - Un *array* di cognomi
 - Un *array* di luoghi di nascita
 - Un *array* di età
 - Un *array* di altezze
 - Un *array* di codici fiscali
- E' però una soluzione pesante e poco leggibile: abbiamo 6 diversi array, mentre quello che vorremmo fare concettualmente è semplicemente rappresentare un solo *array* di **persone**

Soluzione migliore

- Per realizzare una soluzione in cui la struttura dati rappresenti in modo molto più chiaro e semplice i dati del problema, abbiamo bisogno di poter definire direttamente un *tipo di dato persona*
 - Del quale possiamo dire che contiene un nome, un cognome, un luogo di nascita e così via ...
- Tutto questo si può fare in C/C++ mediante il costrutto **struct**, come mostrato nel seguente esempio

Esempio dichiarazione struct

- Dichiarazione del nuovo tipo di dato persona

```
struct persona {  
    char nome[16];  
    char cognome[21];  
    char luogo_nascita[21];  
    int eta;  
    double altezza;  
    char codice_fiscale[17];  
};
```

- Una volta dichiarato il nuovo tipo di dati persona, è possibile definire variabili di tale tipo
- Ad esempio si può scrivere la seguente definizione (solo in C++, in C va ripetuto **struct**, come vedremo in seguito):

```
    persona Mario;
```
- Che cos'è la variabile **Mario**?
 - Una variabile strutturata composta da tre stringhe, un **int**, un **double** ed un'altra stringa

Oggetti di tipo struttura

- Oggetto di tipo struttura
 - ennupla ordinata di elementi, detti **membri** o **campi**, ciascuno dei quali ha un suo nome ed un suo tipo
Esempio: il campo *nome* nel tipo **persona**
- In altri linguaggi il tipo struttura è spesso chiamato **record**
- Un oggetto di tipo struttura differisce da un array per due aspetti:
 - Gli elementi non sono vincolati ad essere tutti dello stesso tipo
 - Ciascun elemento ha un nome

Definizione tipi di dato nuovi

- Mediante il costrutto **struct**, si possono di fatto dichiarare nuovi tipi di dato
 - Ad esempio, il precedente tipo **persona** è un vero e proprio nuovo tipo di dato, che si può utilizzare a sua volta per definire nuovi oggetti di quel tipo
- Per brevità chiameremo semplicemente *tipi struttura* i tipi di dato dichiarati attraverso il costrutto **struct**

- Dichiarazione di un tipo struttura:

```
struct <nome_tipo> { <lista_dichiarazioni_campi> } ;
```

Nome del nuovo tipo

NOTA: come per gli enum si usa il ; dopo una }
Motivo: come vedremo ci potrebbe essere una definizione di variabile/i

- Definizione di oggetti di un tipo strutturato
<nome_tipo>:

```
[const] <nome_tipo>  
    <identificatore1>, <identificatore2>, ... ;
```

Esempio

Nome del nuovo tipo

```
struct frutto {  
    char nome[20];  
    float peso, diametro;  
};
```

Campi

Dichiarazione di due campi

```
frutto f1, f2;
```

Definizione variabili di tipo *frutto*

Definizione contestuale

- Si possono definire degli oggetti di un dato tipo strutturato anche all'atto della dichiarazione del tipo stesso, con la seguente sintassi

```
[const] struct [ <nome_tipo> ]  
    { <lista_dichiarazioni_campi> }  
    <identific_1>, <identific_2>, ... ;
```

- Esempio:

Nome del nuovo tipo (opzionale)

```
struct frutto {  
    char nome[20];  
    float peso, diametro;  
} f1, f2;
```

Campi

Variabili

Selezione campi

- Per selezionare i campi di un oggetto strutturato si utilizza la notazione a punto
`<nome_oggetto>.<nome_campo>`

- Ad esempio, dato

```
struct frutto {  
    char nome[20];  
    float peso, diametro; } f;
```

si può accedere ai campi di f mediante

`f.nome` `f.peso` `f.diametro`

- che risultano essere normali variabili, rispettivamente di tipo stringa e di tipo float
- Esempi:

```
f.peso = 0.34; cout<<f.nome<<endl ;
```

Esempio

```
struct coordinate { int x, y; };
```

```
main()
```

```
{
```

```
    coordinate p1, p2, punto3;
```

```
    p1.x=10;  p1.y=20;  p2.x=30;  p2.y=70;
```

```
    punto3.x = p1.x + p2.x;
```

```
    punto3.y = p1.y + p2.y;
```

```
    cout<<"Coordinate risultanti:"
```

```
        <<" Ascissa="<<punto3.x<<
```

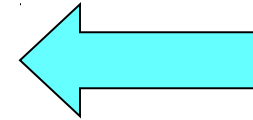
```
        <<" e Ordinata="<<punto3.y<<endl ;
```

```
}
```

- Dalla nona esercitazione:
 - *struttura.cc*

Contenuto lezione

- Stringhe
- Struct
 - Operazioni
 - Progettazione strutture dati e passaggio parametri
- Matrici statiche
 - Implementazione
 - Passaggio alle funzioni



Inizializzazione

- Un oggetto struttura può essere inizializzato
 - elencando i valori iniziali dei campi fra parentesi graffe
 - Esempio:

```
struct coord { int x, y; } ;  
coord p1 = {3, 2} ;
```
 - Copiando il contenuto di un altro oggetto dello stesso tipo
 - Esempio:

```
coord p2 = p1 ;
```
 - Equivale ad una inizializzazione campo per campo
 - Ossia: **`p2.x = p1.x ; p2.y = p1.y ;`**

Assegnamento

- L'assegnamento tra oggetti di tipo struttura equivale ad una copia campo per campo
 - Esempio:
`coord p1 = {3, 2} ;`
`coord p2 ;`
`p2 = p1 ;`
- I due oggetti devono essere dello stesso tipo struttura
- **NON E' CONSENTITO**, invece, fare assegnamenti di oggetti struttura **con nomi di tipi diversi**, anche se i due tipi contenessero gli stessi campi

Esempi di assegnamenti

```
struct coordinata { int x; int y;} p1, p2;
```

```
struct coor { int x; int y;} t1, t2;
```

```
int k;
```

```
. . .
```

```
p2 = p1;
```

```
p1 = t2;
```

```
t2 = t1;
```

```
k = p1;
```

Quali sono validi e quali no?

Risposta

```
struct coordinata { int x; int y;} p1, p2;
```

```
struct coor { int x; int y;} t1, t2;
```

```
int k;
```

```
. . .
```

```
p2 = p1;
```

SI

```
p1 = t2;
```

NO

```
t2 = t1;
```

SI

```
k = p1;
```

NO

Strutture contenenti array

```
struct abc { int x; int v[5]; } ;
```

```
abc p1 = {1, {1, 2, 5, 4, 2}} ;
```

```
abc p2 ;
```

```
p2 = p1 ;
```

A cosa equivale?

```
struct abc { int x; int v[5]; } ;
```

```
abc p1 = {1, {1, 2, 5, 4, 2}} ;
```

```
abc p2 ;
```

```
p2 = p1 ;
```

Equivale a

```
p2.x = p1.x ;
```

```
for (int i = 0 ; i < 5 ; i++)  
    p2.v[i] = p1.v[i] ;
```

Domanda

```
struct abc { int x; int v[5]; } ;
```

```
abc p1 = {1, {1, 2, 5, 4, 2}} ;
```

```
abc p2 ;
```

```
p2.v = p1.v ; // E' corretto?
```

NO

- Esiste quindi un metodo per ottenere la copia tra due *array* senza ricorrere ad un ciclo?

- Sì, basta definire un tipo struttura che contiene semplicemente un array
- Se si effettua l'assegnamento tra due oggetti di tale tipo struttura
 - Si ottiene la copia, elemento per elemento,
 - dell'array contenuto nell'oggetto di origine
 - nell'array contenuto nell'oggetto di destinazione

Uso campi o intero oggetto

```
struct frutto { char nome[20]; float peso, diametro; };  
  
main()  
{  
    frutto f1, f2, f3;  
    float somma;  
    f1.nome = {'m', 'e', 'l', 'a', '\0' }; // ERRATO !!!!!  
    f1.peso=0.26;  
    f2.nome={'a', 'r', 'a', 'n', 'c', 'i', 'a', '\0'};  
                                                // ERRATO !!!!!  
    f2.peso=0.44;  
    somma = f1.peso + f2.peso; ← Utilizzo dei campi  
    f3 = f2; ← Utilizzo dell'intero oggetto  
}
```

- Si può definire un *array* di oggetti di tipo **struct**?

- Ovviamente sì
- Esempio:

```
struct coord { int x, y; } ;  
coord vett[10] ;
```
- Come si accede agli elementi di un array se tali elementi sono di tipo **struct**?
- E come si accede ai singoli campi di un elemento dell'array nel caso in cui tale elemento sia di tipo **struct**?

- Si accede agli elementi con la solita notazione vista finora
- Si accede ai campi di un elemento combinando la notazione per accedere all'elemento con quella per accedere ai campi dell'elemento stesso
- Esempio:

```
struct coord { int x, y; } ;
coord vett[10] ;
// Nella prossima riga assegniamo il valore 2
// al campo x del terzo elemento dell'array
vett[2].x = 2;
vett[2].y = 3 ; // Assegniamo 3 al campo y
cout<<vett[2].x<<endl ; // Stampa campo x
```

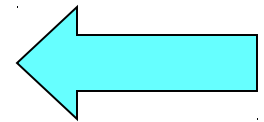
- Quali sono la sintassi e la semantica del passaggio di un array di oggetti di tipo **struct**?

- Le stesse del passaggio di un array di oggetto di tipo primitivo (quale ad esempio `int`)

- Dalla nona esercitazione:
 - *traccia_classifica_solo_elenco.txt*
 - *traccia_classifica.txt*

Contenuto lezione

- Stringhe
- Struct
 - Operazioni
 - Progettazione strutture dati e passaggio parametri
- Matrici statiche
 - Implementazione
 - Passaggio alle funzioni



Esercizio

- Si dichiara una struttura dati in grado di rappresentare l'oggetto astratto *figura piana trapezio*. Definito ed inizializzato un oggetto di tipo trapezio, se ne calcolino il perimetro e l'area

Dati e formule

- Un trapezio è caratterizzato da:
 - Base maggiore (B)
 - Base minore (b)
 - Lato sinistro ($lato_s$)
 - Lato destro ($lato_d$)
 - Altezza (h)
- Per calcolare il perimetro, si applica la formula:
 - $B + b + lato_s + lato_d$
- Per calcolare l'area, si applica la formula:
 - $((B + b) * h) / 2$



Progetto struttura dati 1/2

- Come si rappresentano le informazioni relative ad un trapezio?
 - A basso livello sono dati omogenei (numeri reali), quindi si potrebbe utilizzare un array
 - bisogna ricordare qual è l'indice usato per ciascuno degli elementi del trapezio
 - Pur essendo dati omogenei a basso livello, tali informazioni si riferiscono ad elementi
 - **concettualmente distinti nel dominio del problema**
 - un tipo struttura permette di assegnare un nome distinto a ciascun elemento del trapezio
 - **logicamente correlati** per descrivere un dato trapezio
 - l'uso del tipo struttura permette di **raggruppare in un unico tipo di dato** tale insieme correlato di informazioni

Progetto struttura dati 2/2

- La seconda possibilità è quindi dichiarare un tipo struttura *trapezio*
- Quale scelta è migliore?
- Per rispondere in modo ancora più accurato vediamo il codice che scaturisce dalle due diverse soluzioni

Possibili strutture dati

- array

trapezio[0] ~ Base maggiore
trapezio[1] ~ Base minore
trapezio[2] ~ Lato sinistro
trapezio[3] ~ Lato destro
trapezio[4] ~ Altezza

- tipo struttura

```
struct trapezio_t {  
    double base_maggiore;  
    double base_minore;  
    double lato_s;  
    double lato_d;  
    double h;  
};
```

Programma con array

```
main()
{
    double trapezio[5] = {15, 10, 4, 6, 3};
    double perimetro, area;
    perimetro=trapezio[0]+trapezio[1]+trapezio[2]+trapezio[3];
    area=(trapezio[0]+trapezio[1])*trapezio[4])/2.;
    cout<<"Perimetro="<<perimetro<<" Area="<<area<<endl ;
}
```

Programma con tipo struttura

```
main()
{
    trapezio_t trapezio = {15, 10, 4, 6, 3};
    double perimetro, area;
    perimetro = trapezio.base_maggiore +
                trapezio.base_minore +
                trapezio.lato_s + trapezio.lato_d;
    area = ( trapezio.base_maggiore + trapezio.base_minore )
           * trapezio.altezza / 2;

    cout<<"Perimetro="<<perimetro<<" Area="<<area<<endl ;
}
```


Leggibilità/organizzazione 1/3

- Il precedente esercizio è uno dei casi in cui i tipi strutturati permettono di scrivere codice di maggiore qualità rispetto ad un insieme di array:

1) Maggiore leggibilità delle operazioni

- I campi sono acceduti mediante nomi significativi

2) Migliore organizzazione dei dati

- Informazioni logicamente correlate sono raggruppate assieme nello stesso tipo di dato

Leggibilità/organizzazione 2/3

- 3) Migliore leggibilità delle chiamate di funzione
- In merito, un errore che bisogna evitare di commettere è invocare qualche funzione passando troppi parametri
 - Il numero di parametri oltre il quale un lettore sicuramente si perde è **sette**, ma anche valori non troppo più bassi portano a codice poco leggibile
 - Spesso l'errore nasce dall'usare un errato livello di astrazione: si hanno in pratica funzioni che lavorano **concettualmente** su un dato oggetto, anche se magari non ne leggono/scrivono proprio tutti i campi
 - Ma si commette l'errore di passare uno ad uno tutti e soli i campi interessati
 - La soluzione migliore in questi casi è invece passare **l'intero oggetto**

Leggibilità/organizzazione 3/3

- Se invece la funzione **anche concettualmente** lavora solo su alcuni campi, allora è meglio passare solo tali campi
 - Se la funzione continua ad avere troppi parametri allora a volte l'errore è che la funzione **vuole fare troppe cose** e **va spezzata** in più funzioni più semplici
- D'ora in poi quindi organizziamo opportunamente i dati mediante tipi struttura e cerchiamo sempre di passare parametri alle funzioni al livello di astrazione più appropriato (intero oggetto o singoli campi a seconda dei casi)

Passaggio/ritorno 1/2

- Gli oggetti struttura possono essere passati/ritornati per valore
 - Nel parametro formale finisce la copia campo per campo del parametro attuale
 - Quindi ad esempio sarebbero **copiati tutti** gli elementi di eventuali campi *array*
- Questo può essere molto oneroso
 - Per esempio se l'oggetto contiene un *array* molto molto grande
 - C'è una soluzione alternativa efficiente ???

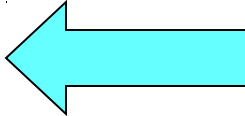
Passaggio/ritorno 2/2

- Passaggio/ritorno per riferimento
- Però, come sappiamo, nel passaggio per riferimento si rischia la modifica indesiderata!
 - Utilizzare quindi, come già visto, il qualificatore **const**

Esercizi d'esame

- Svolgere le prove scritte e di programmazione riportate prima degli esercizi sulle matrici nella nona esercitazione

Contenuto lezione

- Stringhe
- Struct
 - Operazioni
 - Progettazione strutture dati e passaggio parametri
- Matrici statiche 
 - Implementazione
 - Passaggio alle funzioni

Editor grafico ...

- Compiliamo, eseguiamo e giochiamo con
 - *lab9/matrici/disegno.cc*
 - Se ancora non è presente nella nona esercitazione dell'edizione di quest'anno, cercarlo nella stessa esercitazione nella raccolta dell'anno precedente
- Chi ne è l'autore?
 - Voi
 - Dopo aver appreso le prossime nozioni ...

Definizione matrice

- Tabella ordinata di elementi
- Esempio bidimensionale:

a_{ij}

m colonne
 j cresce

n righe
 i cresce

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix}$$

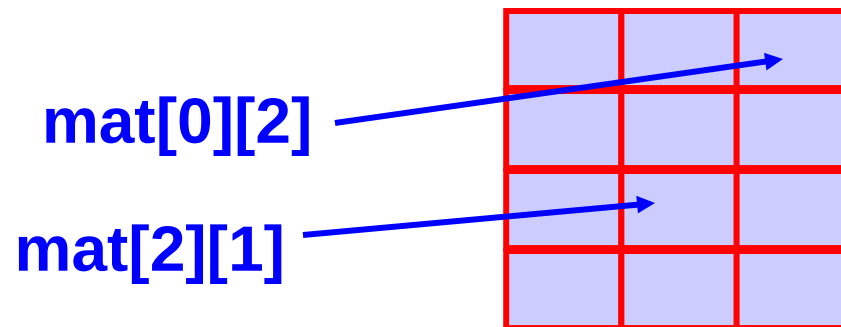
matrice $n \times m$

Matrice bidimens. in C/C++

- SINTASSI della definizione di una variabile o di una costante con nome di tipo **matrice bidimensionale statica**:

```
[const] <tipo_elementi_matrice>  
    <identificatore> [ <espr-costante> ] [ <espr-costante> ] ;
```

- Esempio: matrice di 4x3 oggetti (tutti) di tipo **double**
double mat[4][3] ;



- Dalla decima esercitazione:
 - *riempi_stampa_matrice.cc*

Sintassi matrice k dimensioni

- SINTASSI della definizione di una variabile o di una costante con nome di tipo **matrice statica k-dimensionale**:
[const] *<tipo_elementi_matrice>*
<identificatore> [*<espr-cost_1>*] [*<espr-cost_2>*] ... [*<espr-cost_K>*] ;
ove *<espr-cost_i>* fornisce il numero di elementi dell'*i*-esima dimensione
- Di conseguenza, per accedere ad un elemento bisogna fornire tanti indici quante sono le dimensioni
 - l'*i*-esimo indice può assumere valori compresi fra 0 e (*<espr-cost_i>* - 1)
- In particolare, il generico elemento di una matrice è denotato dal nome della matrice seguito dai valori degli indici racchiusi tra []

Inizializzazione matrici

- Generalizzazione della sintassi per gli *array* monodimensionali

- Esempio:

```
int mat[3][4] = { {2, 4, 1, 3},  
                 {5, 3, 4, 7},  
                 {2, 2, 1, 1} } ;
```

- Il numero di colonne **deve** essere specificato
- Il numero di righe può essere omissso, nel qual caso coincide col numero di righe che si inizializzano
- Elementi non inizializzati hanno valori casuali o nulli a seconda che si tratti di un oggetto locale o globale
- Non si possono inizializzare più elementi di quelli presenti

Esercizio

- Data una matrice di dimensione $M \times M$ di valori reali inizializzata a tempo di scrittura del programma, si calcoli la differenza tra la somma degli elementi della diagonale principale e la somma degli elementi della diagonale secondaria
- Esempio e suggerimenti nelle prossime slide

Esempio indici matrice 5x5

| | | | | |
|-----|-----|-----|-----|-----|
| 0,0 | 0,1 | 0,2 | 0,3 | 0,4 |
| 1,0 | 1,1 | 1,2 | 1,3 | 1,4 |
| 2,0 | 2,1 | 2,2 | 2,3 | 2,4 |
| 3,0 | 3,1 | 3,2 | 3,3 | 3,4 |
| 4,0 | 4,1 | 4,2 | 4,3 | 4,4 |

- Gli elementi della diagonale principale sono caratterizzati dagli indici: $[i][i]$
- Gli elementi della diagonale secondaria sono caratterizzati dagli indici: $[i][M-1-i]$
- Quindi, per scandire tutti gli elementi delle due diagonali è sufficiente un unico ciclo (e quindi un solo indice)

Algoritmo

- Inizializzare due variabili a 0 e sommarvi rispettivamente i valori degli elementi della prima e della seconda diagonale
- Stampare il valore finale della variabile che contiene la differenza tra le due variabili

Struttura dati

- Una costante (int) per denotare la dimensione della matrice: $M=100$
- Una matrice bidimensionale di double pari a $M \times M$
- Un indice (int) per scandire la matrice
- Due variabili ausiliarie (double) per sommarvi i valori delle diagonali

Programma

```
main()
{  const int M=100 ;
   double somma_d1=0., somma_d2=0.;
   double mat[M][M];

   for (int i=0; i<M; i++)
       for(int j=0; j<M; j++)
           cin>>mat[i][j] ;

   for (int i=0; i<M; i++) {
       somma_d1 = somma_d1+mat[i][i];
       somma_d2 = somma_d2+mat[i][M-1-i];
   }
   cout<<"Differenza valori "
        <<somma_d1-somma_d2<<endl;
}
```

Esercizio per casa

- Data una matrice di dimensione $M \times N$ di valori interi, si calcoli il numero complessivo di elementi positivi, negativi e nulli

Algoritmo

- Per scandire tutti gli elementi della matrice possiamo utilizzare due cicli innestati
- Inizializzare due variabili a 0 e sommarvi tutti gli elementi che risultano positivi e negativi
- Serve un'altra variabile per gli elementi nulli?
- Stampa il valore finale delle due o tre variabili

Struttura dati

- Due costanti (int) per denotare la dimensione massima delle righe e delle colonne della matrice:
max_R=100, max_C=1000
- Una matrice bidimensionale di int pari a
max_R * max_C
- Due (o tre ?) variabili ausiliarie (int) come contatori dei valori positivi e negativi

Programma

```
main()
{
    const int max_R = 100, max_C = 1000 ;
    int positivi=0, negativi=0 ;
    int mat[max_R][max_C];

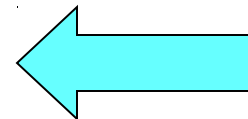
    <si ipotizza che la matrice venga inizializzata in qualche modo>

    for (int i=0; i<max_R; i++)
        { for (int j=0; j<max_C; j++) {
            if (mat[i][j]>0) positivi++;
            else if (mat[i][j]<0) negativi++;
        }
    }

    cout<<"Valori positivi= "<<positivi<<", negativi = "<<negativi
    <<"", nulli = "<< (max_R*max_C - positivi - negativi)
    <<endl ;
}
```

Contenuto lezione

- Stringhe
- Struct
 - Operazioni
 - Progettazione strutture dati e passaggio parametri
- Matrici statiche
 - Implementazione
 - Passaggio alle funzioni



Implementazione matrice

- Considerando la notazione con cui viene definita una matrice e quella con cui si accede ai suoi elementi, forse una matrice è un tipo derivato costruito a partire da un tipo che conosciamo già?

Array di array

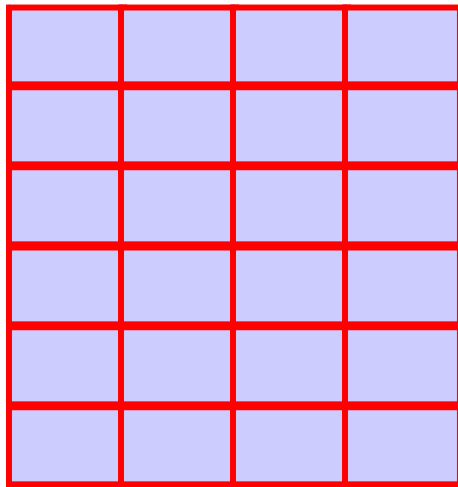
- Sì
- Nel linguaggio C/C++, una matrice è a tutti gli effetti un **array di array**
 - Gli array combinati per ottenere una matrice sono organizzati per righe consecutive
- Ad esempio

```
int mat[M][N] ;
```

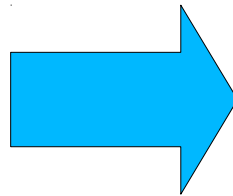
definisce un array di M array da N elementi ciascuno, ossia M righe da N colonne ciascuna, come mostrato nel seguente esempio numerico

Organizzazione matrice

```
int mat[6][4] ;
```



mat[6][4]



mat[0]



mat[1]



mat[2]



mat[3]



mat[4]



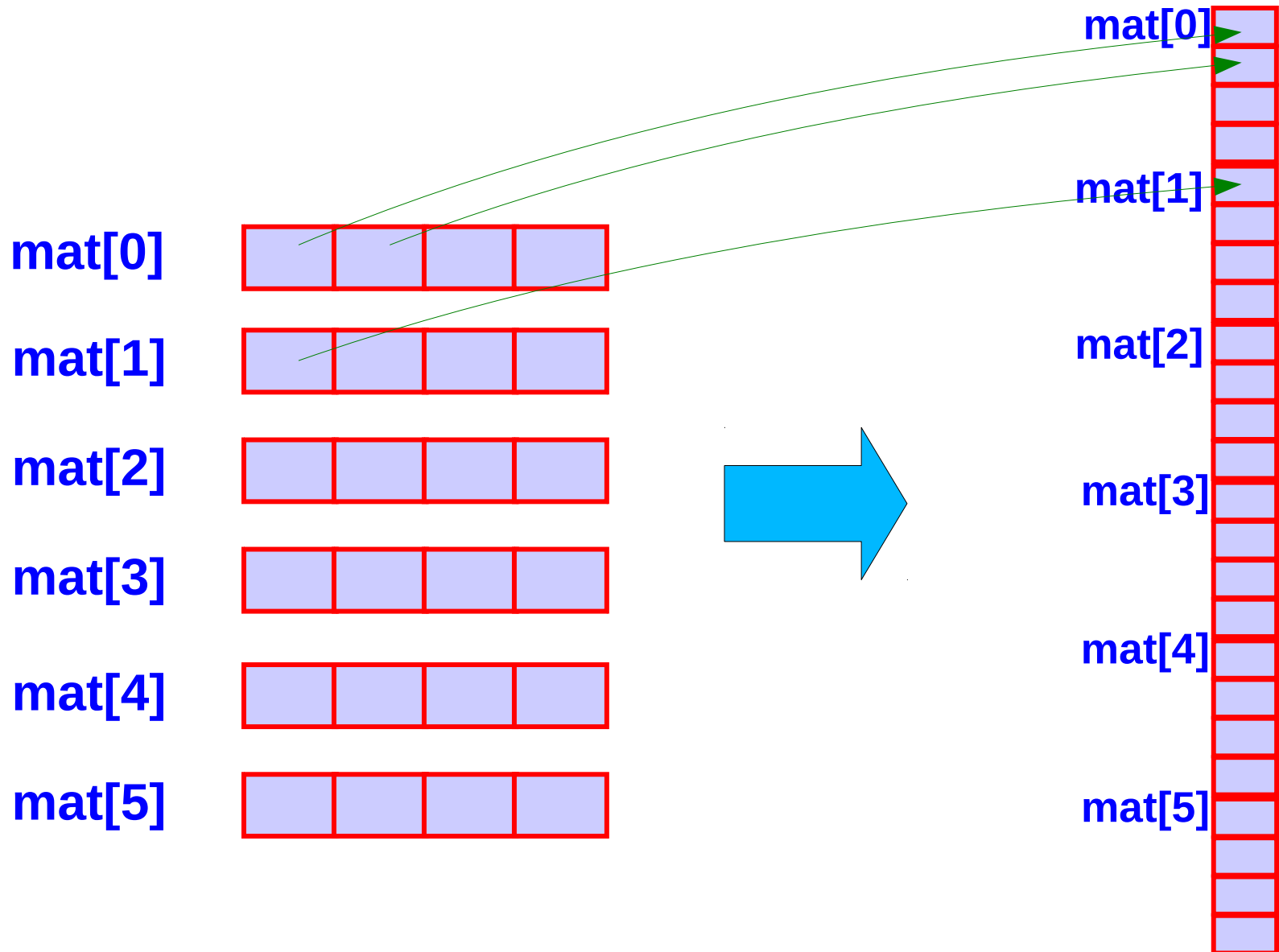
mat[5]



Array di array in memoria 1/2

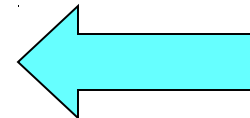
- Siccome un array è una sequenza contigua di elementi in memoria, allora un array di array è una sequenza contigua di array in memoria
- Un esempio è mostrato nella seguente slide

Array di array in memoria 2/2



Contenuto lezione

- Stringhe
- Struct
 - Operazioni
 - Progettazione strutture dati e passaggio parametri
- Matrici statiche
 - Implementazione
 - Passaggio alle funzioni



Passaggio righe matrice 2D 1/2

- Riassumendo quanto detto nelle precedenti slide:
`int mat[M][N] ;`
definisce un array di M array da N elementi ciascuno
- Cos'è quindi `mat[i]` con $i = 0, 1, \dots, M - 1$?

Passaggio righe matrice 2D 2/2

- E' un *array* di N elementi
- Quindi data una matrice di N colonne, come si passa una delle righe ad una funzione che prende in ingresso un *array* lunghezza N ?
- Vediamo con un esercizio: *calcola_somma_righe.cc* della decima esercitazione
- A voi la generalizzazione per il passaggio di fette di matrici con più di due dimensioni

Passaggio matrici

- Così come gli array monodimensionali, gli array di array sono **passati per riferimento**
- La dichiarazione/definizione di un parametro formale di tipo matrice bidimensionale è la seguente:
[const] *<tipo_elementi>*
<identificatore> **[][<numero_colonne>]**
 - Nessuna indicazione del numero di righe !!!
 - La funzione pertanto non conosce implicitamente il numero di righe della matrice
 - Se presente, il qualificatore **const** fa sì che la matrice non sia modificabile
- Nell'invocazione della funzione, una matrice si passa **scrivendone semplicemente il nome**

Esempio

```
const int num_col = 4 ;
```

```
void fun(int mat[][num_col], int num_righe) ;
```

```
main()
```

```
{  
    const int M = 3 ;  
    int A[M][num_col] ;  
    fun(A, M) ;  
    ...  
}
```

Domanda

- Di quale informazione ha bisogno il compilatore per poter generare il codice che accede al generico elemento di una matrice?

- Dell'indirizzo di tale elemento in memoria

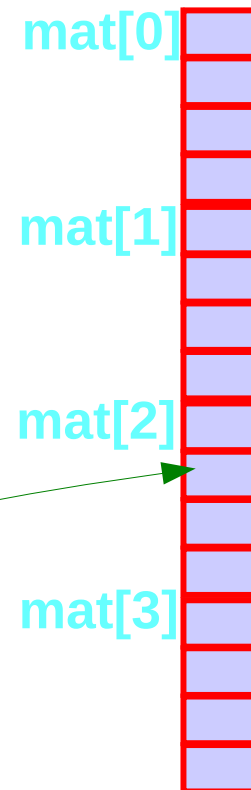
Domanda

- Di quali informazioni ha bisogno per calcolare tale indirizzo?
 - Ricordate che le righe sono memorizzate l'una dopo l'altra

Risposta

- 1) La prima locazione in cui è memorizzata la matrice
- 2) Le dimensioni di ciascun elemento (numero di byte occupate, dipende dal tipo degli elementi)
- 3) La lunghezza di ciascuna riga, ossia il numero di colonne della matrice

- Tali informazioni servono per accedere, ad esempio, al secondo elemento della terza riga di questa matrice, ossia l'elemento di indici [2][1]
- L'indirizzo dell'elemento è infatti dato da:
indirizzo_matrice +
dim_elementi*(lun_riga*2 + 1)



Indirizzo generico elemento

- In generale l'indirizzo del generico elemento di indici i e j è dato da

$\text{indirizzo_matrice} + \text{dim_elementi} * (\text{lun_riga} * i + j)$

Domanda

- Alla luce di quanto abbiamo appena capito, come mai è obbligatorio passare il numero di colonne?

Accesso elementi 1/2

- Perché, come visto, è necessario conoscere il numero di colonne di una matrice bidimensionale per calcolare l'indirizzo di un suo generico elemento

Array di stringhe

- Per analogia con quanto detto in precedenza, un array di stringhe si realizza mediante una matrice di tipo **char**
- Esempio: Elenco dei nomi dei giorni della settimana:

```
char giorni[7][11] =  
{ "lunedì", "martedì", "mercoledì",  
  "giovedì", "venerdì", "sabato",  
  "domenica" } ;
```

| | | | | | | | | | | |
|---|---|---|---|---|---|----|----|----|---|----|
| l | u | n | e | d | i | ' | \0 | | | |
| m | a | r | t | e | d | i | ' | \0 | | |
| m | e | r | c | o | l | e | d | i | ' | \0 |
| g | i | o | v | e | d | i | ' | \0 | | |
| v | e | n | e | r | d | i | ' | \0 | | |
| s | a | b | a | t | o | \0 | | | | |
| d | o | m | e | n | i | c | a | \0 | | |

- Dalla nona esercitazione:
 - *disegno.cc*

Esercizi per casa

- Seguono degli esercizi sulle matrici per casa

Battaglia navale semplificata

- Realizzare un programma che, dopo aver fatto creare una mappa 10x10 con 12 navi da 1 cella in posizioni casuali, consenta ad un giocatore di “scoprire” tutte le posizioni delle nave avversaria.
- La classifica dei record viene mantenuta rispetto al numero dei colpi necessari per scoprire tutte le nave nemiche.
- Estensione: si visualizzi la mappa, con la posizione delle navi scoperte, i tiri effettuati andati a vuoto, e quelli andati a buon fine
- Per implementare bene il programma partire dalla realizzazione delle seguenti funzioni propedeutiche

Funzioni propedeutiche

- Scrivere una funzione INSERT che riceva in input un numero di navi e le inserisca casualmente in una mappa di dimensioni 10x10
 - Si assuma che ciascuna nave occupi 1 cella
 - Si faccia attenzione a non posizionare le navi in celle coincidenti
- Scrivere una funzione TIRO che riceva in input una coordinata (ovvero due elementi interi), e restituisca se il tiro ha colpito o meno una nave

Battaglia navale

- Scrivere un programma che
 - crei una mappa con le seguenti navi in posizioni casuali:
 - 1 nave da 4 celle, 2 navi da 3 celle, 3 navi da 2 celle, 4 navi da 1 cella (scegliere a proprio piacimento le dimensioni della mappa)
 - consenta ad un giocatore di “scoprire” le posizioni delle navi avversarie
 - Mantenga una classifica dei record rispetto al numero dei colpi necessari per scoprire tutte le navi nemiche
- Estensione: si visualizzi la mappa, con la posizione delle navi scoperte, i tiri effettuati andati a vuoto, e quelli andati a buon fine
- Per implementare bene il programma partire dalla realizzazione delle seguenti funzioni propedeutiche

Funzioni propedeutiche

- Data una mappa di dimensione $M \times M$, si inseriscano casualmente (in posizioni non sovrapposte):
 - 1 nave da 4 celle
 - 2 navi da 3 celle
 - 3 navi da 2 celle
 - 4 navi da 1 cella
- Si accettano navi in diagonale?
- Scrivere poi una funzione che, presa in ingresso una coordinata, stampi su video se il tiro ha colpito o meno una nave

Gioco della vita 1/2

- Una mappa di dimensione $N \times M$ rappresenta il mondo. Ogni cella può essere occupata o meno da un organismo. Partendo da una configurazione iniziale di organismi, questa popolazione evolve nel tempo secondo tre regole genetiche:
 - un organismo sopravvive fino alla generazione successiva se ha 2 o 3 vicini;
 - un organismo muore, lasciando la cella vuota, se ha più di 3 o meno di 2 vicini;
 - ogni cella vuota con 3 vicini diventa una cella di nascita e alla generazione successiva viene occupata da un organismo.
- Si visualizzi l'evoluzione della popolazione nel tempo

Gioco della vita 2/2

- Nota

Il concetto di “vicinanza” in una tabella raffigurante il mondo può essere interpretato in 2 modi:

- Al di là dei bordi c'è il vuoto che non influenza il gioco, per cui ci sono punti interni che hanno 8 potenziali “vicini”, punti sulle righe e colonne estreme che hanno 5 “vicini”, punti ai vertici che hanno 3 “vicini”
- I bordi estremi confinano tra di loro: la colonna “0” è “vicina” alla colonna “M-1”, così come la riga “0” è “vicina” alla riga “N-1” (con attenzione a trattare i vertici!)

Esercizi d'esame

- Terminare la nona esercitazione