Lezione 11

Laura Bozzelli

a.a. 2020/2021

Sommario - Lezione 11: Tipi di dati definiti da utente

- Strutture.
- Unioni.
- Tipi enumerativi.
- Introduzione agli array.
- Array monodimensionali.
- Passaggio per riferimento degli array.

Tipi definiti da utente: strutture

- Nel linguaggio C, una **struttura**, detta anche aggregato, è una collezione di dati (variabili), di uno o più tipi, raggruppate sotto un unico nome.

- Le strutture possono contenere variabili di tipi (predefiniti o definiti dall'utente) differenti.

- Aiutano ad organizzare dati complessi in quanto consentono di trattare come un unico oggetto (dato aggregato) un insieme di variabili correlate.

- Esempio: i dati relativi ad uno studente (nome, cognome, numero di immatricolazione, media esami, ecc..) possono essere raggruppati in un'unica struttura.

Definizione di un tipo struttura (1/5)

Sintassi

```
struct <nome tipo>
{
   <dichiarazione variabile 1>
      ....
   <dichiarazione variabile N>
};
```

Dichiarazioni di variabili racchiuse tra parentesi graffe, terminante con un punto e virgola, e avente come intestazione la parola chiave **struct** seguita da un identificatore di tipo.

- **<nome tipo>**: è qualsiasi identificatore valido (come visto per gli identificatori delle variabili semplici o dei nomi di funzione). Chiamato anche **tag**, rappresenta l'**identificatore del tipo di struttura**. Similmente agli identificatori (parole chiavi) dei tipi predefiniti (int, char, float,ecc..), esso viene utilizzato per dichiarare variabili del dato tipo.

Definizione di un tipo struttura (2/5)

Sintassi

```
struct <nome tipo>
{
    <dichiarazione variabile 1>
        ....
     <dichiarazione variabile N>
};
```

- **<dichiarazione variabile 1>** …. **<dichiarazione variabile N>** : dichiarazioni di variabili (di tipo predefinito o definito dall'utente) senza inizializzazione e senza specificatori di classi di memoria.
- Le variabili dichiarate entro le parentesi nella definizione di un tipo struttura sono dette **membri** o **campi** della struttura.

Buona norma: far rientrare a destra il testo delle dichiarazioni di variabili con un certo livello di indentazione all'interno delle parentesi graffe che lo delimitano.

Definizione di un tipo struttura (3/5)

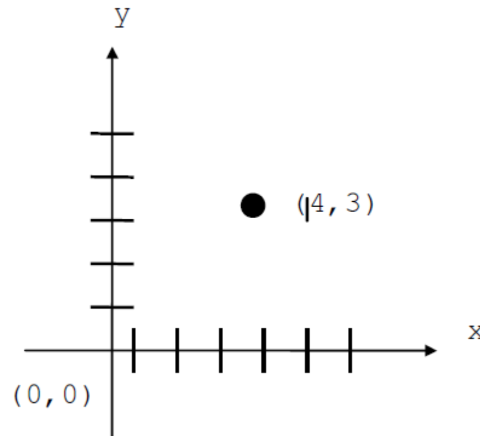
Sintassi

```
struct <nome tipo>
{
   <dichiarazione variabile 1>
       ....
    <dichiarazione variabile N>
};
```

- I **membri** di un tipo di struttura devono avere nomi differenti, ma due tipi di strutture differenti possono contenere membri dello stesso nome senza che ciò crei un conflitto.
- Ogni definizione di un tipo struttura deve terminare con un punto e virgola.
- Una definizione di tipo struttura definisce **un tipo.** Non viene allocata alcuna memoria in corrispondenza di una definizione di tipo. La memoria viene allocata solo tramite dichiarazioni (alias definizioni) di variabili di quel tipo.

Definizione di un tipo struttura (4/5)

Esempio: un qualsiasi punto del piano cartesiano è una coppia ordinata di numeri reali.



Possiamo utilizzare il seguente tipo di struttura per rappresentare tali punti.

```
struct Punto
{
    float x; //coordinata lungo l'asse X
    float y; //coordinata lungo l'asse Y
};
```

Definizione di un tipo struttura (5/5)

- Una definizione di tipo struttura può comparire nel corpo di una funzione. In questo caso, il suo **campo di azione o scope** va dal punto in cui tale definizione è scritta al termine del corpo della funzione.
- Quando una definizione di tipo struttura compare all’esterno di una definizione di funzione, il suo **campo di azione o scope** va dal punto della definizione fino alla fine del file in cui la definizione si trova. Per consentire l’accesso in più file sorgenti, conviene porre la definizione di un tipo struttura in un file header ed includere il file tramite la direttiva **#include** del preprocessore.

Dichiarazione variabili di tipo struttura (1/3)

Sintassi per tipi di struttura con identificatore di tipo:

struct <nome tipo struttura> <nome var 1>, …, <nome var N>;

- **<nome tipo struttura>**: identificatore della definizione di tipo struttura.
- **<nome var 1>**, .., **<nome var N>**: nomi di variabili (qualsiasi identificatore valido).

La dichiarazione di variabile alloca una area contigua di memoria per contenere tutti i **membri** di quella variabile.

Esempio:

```
struct Punto
{
    float x; //coordinata lungo l'asse X
    float y; //coordinata lungo l'asse Y
};
```

```
struct Punto punto1, punto2;
struct Punto punto3;
```

Dichiarazione variabili di tipo struttura (2/3)

Variabili di tipo struttura possono essere anche dichiarate immediatamente dopo la definizione dell'associato tipo struttura e prima del terminatore punto e virgola.

```
struct Punto
{
    float x; //coordinata lungo l'asse X
    float y; //coordinata lungo l'asse Y
} punto1, punto2;
```

Dichiarazione variabili di tipo struttura (3/3)

Una definizione di tipo struttura può essere anche **anonima** e, cioè, non contenere alcun identificatore di tipo. In questo caso, le variabili di quel tipo possono essere dichiarate solo immediatamente dopo la definizione di tipo struttura.

Esempio:

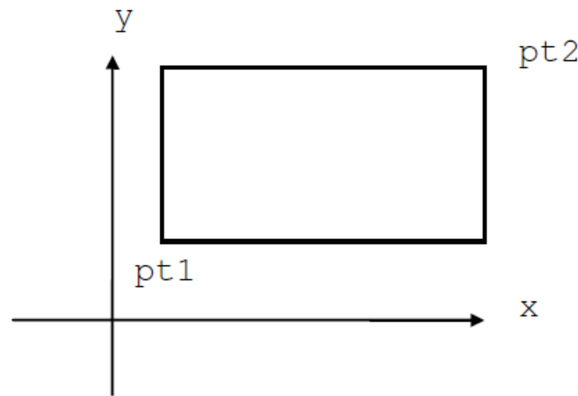
```
struct
{
    float x; //coordinata lungo l'asse X
    float y; //coordinata lungo l'asse Y
} punto1, punto2;
```

NOTA: una definizione di tipo struttura **anonima** può essere parte di una definizione di struttura con identificativo per dichiarare campi di tipo struttura con tipo anonimo.

Strutture annidate

Membri di una struttura possono essere a loro volta variabili di tipo struttura (e, cioè, le strutture possono essere arbitrariamente annidate).

Esempio 2: un rettangolo nel piano cartesiano può essere identificato da due punti, il punto corrispondente all'angolo superiore-destro ed il punto corrispondente all'angolo inferiore-sinistro.



Possiamo utilizzare il seguente tipo di struttura per rappresentare rettangoli nel piano.

```
struct  Rettangolo
{
    struct Punto pt1; //angolo inferiore sinistro
    struct Punto pt2; //angolo superiore destro
};
```

Inizializzazione variabili di tipo struttura (1/2)

Dichiarazione variabile di tipo struttura con inizializzazione:

struct <nome tipo struttura> <nome var> = <inizializzazione>;

Sintassi **<inizializzazione>**: rappresenta la parte di inizializzazione.

{ <Init 1>, ….,<Init N>}

- N deve essere non superiore al numero dei membri della struttura.
- **<Init i>** inizializza l'i-esimo membro ($1 \leq i \leq N$): per un membro di tipo numerico (semplice), è un'espressione numerica generica. Per un membro di tipo struttura è a sua volta un'espressione di inizializzazione oppure il nome di una variabile dello stesso tipo, oppure una chiamata a funzione restituente un dato di quel tipo.
- Se N è inferiore al numero di membri della struttura, i restanti membri sono inizializzati a zero (per membri di tipo numerico).
- Per **dichiarazioni di variabili globali o statiche**, le espressioni numeriche utilizzate devono essere costanti (i valori di inzializzazione devono essere noti a tempo di compilazione).

Inizializzazione variabili di tipo struttura (2/2)

```
struct  Punto
{
    float x; //coordinata lungo l'asse X
    float y; //coordinata lungo l'asse Y
};

struct  Rettangolo
{
    struct Punto pt1; //angolo inferiore sinistro
    struct Punto pt2; //angolo superiore destro
};

struct Rettangolo rett={{1.34,2},{2.34,5.1}};

struct Punto RitornaPunto();//Prototipo di funzione c
                            //che restituisce una struttura Punto

void Prova()
{
    float x = 3.5, y = 2.7;
    struct Punto p1 = {x,y+1};
    struct Rettangolo ret = {p1, RitornaPunto()};
    ......
}
```

Utilizzo dati di tipo struttura (1/3)

Le sole operazioni valide eseguibili su strutture sono le seguenti:

- Assegnazione a variabili di tipo struttura.

  <nome variabile> = <espressione>;

  - dove **<espressione>** può essere **o** un nome di variabile struttura dello stesso di tipo di **<nome variabile>** **oppure** una chiamata di funzione che restituisce come risultato una struttura dello stesso tipo di **<nome variabile>** .
  - Il 'valore aggregato' dell'espressione viene assegnato alla variabile struttura a sinistra dell'istruzione di assegnazione.

```
struct Punto RitornaPunto();//Prototipo di funzione c
                            //che restituisce una struttura Punto

void Prova()
{
    struct Punto p1,p2;
    p1 = RitornaPunto();
    p2 = p1;
}
```

Utilizzo dati di tipo struttura (2/3)

- Passaggio di argomenti di tipo struttura alle funzioni e la restituzione di valori struttura dalle funzioni. Le funzioni possono avere come parametri formali variabili sia di tipo predefinito che di tipo definito dall'utente e restituire valori di tipo arbitrario. Il passaggio degli argomenti è sempre per valore: il valore dell'argomento di tipo struttura viene copiato nel parametro formale di tipo struttura.

Esempio:

```
//Prototipo di funzione avente strutture come parametri formali
//che restituisce una struttura
struct Punto SommaPunti(struct Punto p1,struct Punto p2);
```

- Puntatori a strutture (lo vedremo nel seguito).

Utilizzo dati di tipo struttura (3/3)

- Accesso ai campi (membri) di una variabile di tipo struttura tramite l'**operatore di campo di struttura '.'**.

  <nome variabile>.<nome campo>

  - identifica la locazione di memoria dell'associato campo della variabile struttura. Se tale campo è a sua volta una struttura, si può far seguire all'espressione precedente l'operatore **'.'** per accedere in modo annidato ai suoi membri.

```
struct  Rettangolo
{
    struct Punto pt1; //angolo inferiore sinistro
    struct Punto pt2; //angolo superiore destro
};

float CalcolaArea(struct Rettangolo rect)
{
    return (rect.pt2.y-rect.pt1.y) * (rect.pt2.x-rect.pt1.x);
}
```

Esempi utilizzo dati di tipo struttura

```
struct  Punto
{
    float x; //coordinata lungo l'asse X
    float y; //coordinata lungo l'asse Y
};


struct Punto CreaPunto(float x,float y)
{
    struct Punto temp;
    temp.x=x;
    temp.y=y; //Non esiste conflitto fra i nomi dei parametri e quelli
              //della struttura; al contrario l'omonimia evidenzia la
              //relazione tra essi.
    return temp;
}

struct Punto AggiungiPunti(struct Punto p1, struct Punto p2)
{
    //Il passaggio di argomenti è sempre per valore.
    //La modifica del parametro formale p1 non si ripercuote
    // sull'argomento strutturato passato in input dal chiamante
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
```

Tipi definiti da utente: unioni

- Nel linguaggio C, un’**unione** è una collezione di dati (**membri** o **campi**) che *condividono lo stesso spazio di memoria*.

- Per diverse situazioni in un programma, alcune variabili possono non essere rilevanti mentre altre possono esserlo, e così un’unione fa condividere spazio di memoria. Come le strutture, i campi di un’unione possono essere di tipo diverso. Il numero di byte necessario per memorizzare un’unione deve essere sufficiente a contenere il campo più grande.

- Si può far riferimento soltanto ad un campo alla volta e di conseguenza ad un solo tipo di dati alla volta. È responsabilità del programmatore tenere traccia del campo attualmente utilizzato.

Utilizzo di tipi unione

- La definizione di un tipo unione è analoga alla definizione di un tipo struttura con l'unica differenza che viene utilizzata la parola chiave **union** invece di **struct**.

- Stesso discorso si applica alle dichiarazioni di variabili. C'è, comunque, un'importante differenza nelle dichiarazioni di variabili con inizializzazione. Per le unioni, è **possibile inizializzare solo il primo campo**.

- Similmente alle strutture, le operazioni che è possibile effettuare sulle unioni sono:
  - Assegnazione a variabili di tipo unione.
  - Passaggio di argomenti di tipo unione alle funzioni e la restituzione di valori unione dalle funzioni.
  - Puntatori a unione (lo vedremo nel seguito).
  - Accesso ai campi (membri) di una variabile di tipo unione tramite l'**operatore di campo '.'**.

Esempi utilizzo dati di tipo unione (1/2)

Assumiamo che un oggetto grafico possa essere o un punto o un rettangolo. Utilizziamo una struttura con due campi per rappresentare un oggetto grafico: il primo è un flag indicante se l'oggetto è un punto o un rettangolo. Il secondo campo è un'unione avente due campi: un campo di tipo **Rettangolo** e un campo di tipo **Punto**.

```
struct OggettoGrafico
{
    // è maggiore di zero se
    //l'oggetto grafico è un rettangolo
    int flagRettangolo;

    //Dichiarazione di campo union con
    // definizione anonima di tipo unione
    union
    {
      struct Punto pt;
      struct Rettangolo rect;
    } oggetto;

};
```

Esempi utilizzo dati di tipo unione (2/2)

```c
struct OggettoGrafico
{
    // è maggiore di zero se
    //l'oggetto grafico è un rettangolo
    int flagRettangolo;

    //Dichiarazione di campo union con
    // definizione anonima di tipo unione
    union
    {
      struct Punto pt;
      struct Rettangolo rect;
    } oggetto;

};

void DisegnaRettangolo(struct Rettangolo rect);
void DisegnaPunto(struct Punto pt);

void DisegnaOggettoGrafico(struct OggettoGrafico obj)
{
    if(obj.flagRettangolo)
        DisegnaRettangolo(obj.oggetto.rect);
    else
        DisegnaPunto(obj.oggetto.pt);
}
```

Tipi definiti da utente: enumerazione

- Nel linguaggio C, un'**enumerazione**, è un insieme finito di valori interi (**costanti intere di enumerazione**) rappresentati da nomi simbolici (identificatori).
- Le enumerazioni sono un mezzo efficiente per associare valori interi costanti a dei nomi.

Definizione di un tipo enumerativo (1/3)

Sintassi

```
enum <nome tipo>
{
    <dichiarazione 1>,
    <dichiarazione 2>,
        ....
     <dichiarazione N>
};
```

Dichiarazione di identificatori (nomi simbolici per costanti intere) racchiuse tra parentesi graffe, terminante con un punto e virgola, e avente come intestazione la parola chiave **enum** seguita da un **identificatore di tipo**.

<dichiarazione i> ($1 \leq i \leq N$): è

- o un identificatore valido,
- oppure un'espressione della forma **<nome> = <costante>** dove nome è un identificatore valido e **<costante>** è una costante intera (valore costante dell'identificatore).

Definizione di un tipo enumerativo (2/3)

Sintassi

```
enum <nome tipo>
{
    <dichiarazione 1>,
    <dichiarazione 2>,
        ....
     <dichiarazione N>
};
```

- Gli identificatori nelle varie dichiarazioni devono essere distinti e devono essere unici all'interno del programma.
- Ad identificatori distinti può essere associata la stessa costante intera.
- Se la prima dichiarazione non ha un valore esplicito, al primo identificatore viene assegnato un valore 0.
- Per una dichiarazione distinta dalla prima il cui valore non è esplicitamente assegnato, il valore del nome è ottenuto incrementando di 1 il valore del nome precedente.

Definizione di un tipo enumerativo (3/3)

Esempi:

```
//Assegna il nome FALSE al valore 0
// e il nome TRUE al valore 1
enum Bool { FALSE, TRUE };


//Mesi dell'anno: gli identificatori sono impostati,
//rispettivamente con gli interi da 1 a 12
enum Mesi {GEN=1, FEB,MARZ, APR,MAG,GIU, LUG, AGO, SETT, OTT, NOV,DIC};
```

- Per il campo d’azione o scope di una definizione di un tipo enumerativo valgono le stesse regole viste per le definizioni di tipo struttura.
- Per consentire l’accesso in più file sorgenti di una definizione di tipo, conviene porre la definizione in un file header ed includere il file tramite la direttiva **#include** del preprocessore.

Dichiarazione variabili di tipo enumerativo (1/2)

Sintassi senza inizializzazione:

enum <nome tipo> <var 1>, …, <var N>;

Sintassi con inizializzazione:

enum <nome tipo> <var 1> = <id 1>, …, <var N> =<id N>;

- **<nome tipo >**: identificatore della definizione di tipo enumerativo.
- **<var 1>**, .., **<var N>**: nomi di variabili (qualsiasi identificatore valido).
- **<id 1>**, .., **<id N>**: nomi simbolici per le costanti enumerative utilizzate nella definizione di tipo.

Esempio:

```
//Mesi dell'anno: gli identificatori sono impostati,
//rispettivamente con gli interi da 1 a 12
enum Mesi {GEN=1, FEB,MARZ, APR,MAG,GIU, LUG, AGO, SETT, OTT, NOV,DIC};

enum Mesi meseNascita = FEB;
enum Mesi mese1, mese2;
```

Dichiarazione variabili di tipo enumerativo (2/2)

Variabili di tipo enumerativo (come per i tipi struttura e unione) possono essere anche dichiarate immediatamente dopo la definizione dell'associato tipo enumerativo e prima del terminatore punto e virgola.

```
//Mesi dell'anno: gli identificatori sono impostati,
//rispettivamente con gli interi da 1 a 12
enum Mesi {GEN=1, FEB,MARZ, APR,MAG,GIU, LUG,
            AGO, SETT, OTT, NOV,DIC} mese1 =GIU, mese2 = LUG;
```

Variabili di tipo enumerativo sono variabili numeriche e dunque possono essere utilizzate per comporre espressioni numeriche.

Esempi utilizzo dati di tipo enumerativo

```
enum TipoGrafico{RETTANGOLO, PUNTO};
struct OggettoGrafico
{
    // tipo dell'ggetto grafico
    enum TipoGrafico tipo;

    //Dichiarazione di campo union con
    // definizione anonima di tipo unione
    union
    {
        struct Punto pt;
        struct Rettangolo rect;
    } oggetto;

};

void DisegnaRettangolo(struct Rettangolo rect);
void DisegnaPunto(struct Punto pt);

void DisegnaOggettoGrafico(struct OggettoGrafico obj)
{
    switch(obj.tipo)
    {
        case RETTANGOLO:
            DisegnaRettangolo(obj.oggetto.rect);
            break;
        case PUNTO:
            DisegnaPunto(obj.oggetto.pt);
    }
}
```

Tipi definiti da utente: array (1/2)

- Nel linguaggio C, un **array**, detto anche **vettore**, è una sequenza ordinata di dati dello stesso tipo, raggruppati sotto un unico nome.

- I singoli dati in un array sono chiamati **elementi dell’array** e possono essere acceduti tramite l’**operatore di indicizzazione []**.

- Elementi consecutivi in un array hanno **locazioni di memoria** contigue.

- Il numero di elementi di un array è chiamato **lunghezza dell’array**. Gli array sono in generale dati **a lunghezza variabile** dal momento che la lunghezza di un array può essere specificata tramite espressioni numeriche intere il cui valore è noto a tempo di esecuzione.

Tipi definiti da utente: array (2/2)

- A differenza degli aggregati (strutture e unioni) e dei tipi enumerativi, per definire una variabile di tipo array non bisogna prima definire un tipo array.

- Nella dichiarazione di una variabile di tipo array si specifica il nome della variabile, il tipo comune degli elementi di un array, e la lunghezza dell'array.

- **IMPORTANTE:** a differenza delle variabili semplici (numeriche) e delle variabili di tipo aggregato (struttura o unione) o di tipo enumerativo, una variabile array è una **variabile puntatore** dal momento che il suo valore rappresenta l'indirizzo di memoria della locazione di memoria associata al primo elemento dell'array.

Dichiarazione variabili array (1/2)

Sintassi per array monodimensionali:

<tipo elemento> <nome array> [<lunghezza array>];

- **<tipo elemento>**: rappresenta il tipo comune degli elementi. Per elementi semplici è l'insieme di parole chiave per identificare il tipo semplice (int, double, char, long double, ecc). Per i tipi struttura (risp., unione, risp., enumerazione) è la parola chiave **struct** (risp., **union**, risp., **enum**) seguita dal nome del tipo (come specificato nella definizione di tipo).
- **<nome array>**: nome dell'array (qualsiasi identificatore valido).
- **<lunghezza array>**: rappresenta la lunghezza dell'array. Può essere una qualsiasi espressione numerica di tipo intero.

Esempio: array di strutture

```
struct  Punto
{
    float x; //coordinata lungo l'asse X
    float y; //coordinata lungo l'asse Y
};

//Array di nome traiettoria consistente di 10
//elementi di tipo Punto
struct Punto traiettoria[10];
```

Dichiarazione variabili array (2/2)

Sintassi per array monodimensionali:

<tipo elemento> <nome array> [<lunghezza array>];

- Per dichiarazioni di (variabili) array globali o locali statiche (e, cioè, il cui campo di memoria è statico), la lunghezza dell'array **<lunghezza array>** deve essere una espressione numerica intera **costante** il cui valore, determinabile a tempo di compilazione, deve essere un intero positivo. Altrimenti, il compilatore segnala un errore di sintassi. Per tali variabili, l'area di memoria viene allocata all'inizio dell'esecuzione del programma.
- Per dichiarazioni di variabili array locali automatiche, è possibile utilizzare espressioni numeriche intere generiche per specificare la lunghezza di un array. La valutazione a tempo di esecuzione dell'espressione numerica per la lunghezza dell'array (e la conseguente allocazione dinamica di un'area di memoria per memorizzare l'array) può generare un errore irreversibile se il valore ottenuto non è positivo e eccede la capacità di memoria a disposizione.

Inizializzazione variabili array (1/3)

Dichiarazione array monodimensionale con inizializzazione:

<tipo elemento> <nome array> [<lunghezza array>] =

<inizializzazione>;

<lunghezza array> deve essere un'**espressione intera costante.**

Sintassi **<inizializzazione>**: rappresenta la parte di inizializzazione.

{ <Init 1>, ….,<Init N>}

- N deve essere non superiore alla lunghezza dell'array.
- **<Init i>** inizializza l'i-esimo membro (1≤ i≤N): per elementi semplici, è <u>un'espressione numerica generica</u>. Per elementi aggregati o di tipo enumerativo, è a sua volta <u>un'espressione di inizializzazione</u> oppure il nome di una variabile dello stesso tipo, oppure una chiamata a funzione restituente un dato di quel tipo.

Inizializzazione variabili array (2/3)

Dichiarazione array monodimensionale con inizializzazione:

<tipo elemento> <nome array> [<lunghezza array>] =
<inizializzazione>;

<lunghezza array> deve essere un'**espressione intera costante.**

Sintassi **<inizializzazione>**: rappresenta la parte di inizializzazione.

{ <Init 1>, ….,<Init N>}

- Se N è inferiore al numero di elementi dell'array, i restanti elementi sono inizializzati a zero (per elementi di tipo numerico).
- Per **dichiarazioni di variabili globali o statiche**, le espressioni numeriche utilizzate devono essere costanti (i valori di inizializzazione devono essere noti a tempo di compilazione).

Inizializzazione variabili array (3/3)

```
struct  Punto
{
    float x; //coordinata lungo l'asse X
    float y; //coordinata lungo l'asse Y
};

int cifre[10] ={0,1,2,3,4,5,6,7,8,9};

//Per il primo punto della traiettoria,
//l'ascissa x è inizializzata a 1.6 e
//e l'ordinata a 7.8. Per tutti gli
// altri punti, l'ascissa e l'ordinata sono
//inizializzati a zero.
struct Punto traiettoria[11] = {{1.6,7.8}};
```

Accesso agli elementi di un array (1/4)

- Ad ogni elemento di un array è associato un numero di posizione (**indice**) che va da 0 alla lunghezza dell'array meno 1.
- Si può far riferimento a uno qualunque degli elementi di un array monodimensionale fornendo il nome dell'array seguito dal numero di posizione (indice) dell'elemento racchiuso tra parentesi quadre.

Accesso agli elementi di un array (2/4)

Sintassi accesso agli elementi di un array monodimensionale.

<nome array>[<indice elemento>]

- <indice elemento> è una generica espressione numerica intera.
- L'espressione <nome array>[<indice elemento>] può essere utilizzata alla stessa stregua di una variabile avente come tipo il tipo degli elementi di un array. In particolare, può comparire come operando sinistro in un'istruzione di assegnazione e come parte di un'espressione numerica se il tipo degli elementi è numerico.
- Le parentesi quadre usate per racchiudere l'indice di un array sono considerate in C un operatore. L'**operatore di indicizzazione [ ]** ha la stessa precedenza dell'**operatore ()** usato nelle chiamate di funzione (e, cioè, le parentesi tonde poste dopo il nome della funzione).

```
int A[10];
int i=3;
A[i] = i*i;
```

Accesso agli elementi di un array (3/4)

- È importante assicurarsi che ogni indice che si usa per accedere a un elemento di un array sia dentro i confini dell'array (e, cioè vada da 0 alla lunghezza dell'array meno 1).
- Il C *non fornisce alcun controllo automatico dei confini per gli array* (ciò è compito del programmatore).
- Consentire ai programmi di leggere e scrivere negli elementi di un array al di fuori dei confini dell' array rappresenta un comune difetto di sicurezza.
  - Leggere al di fuori dei confini può fare arrestare il programma o perfino dare l'impressione che questo venga eseguito correttamente, mentre invece usa dati non validi.
  - Scrivere al di fuori dei confini (**overflow del buffer**) può corrompere i dati in memoria di un programma e arrestare il programma.

Accesso agli elementi di un array (4/4)

Inizializzazione degli elementi di un array monodimensionale tramite ciclo controllato da contatore.

Esempio:

```
void Prova(struct Punto pt, int size)
{
    if(size >0)
    {
        struct Punto A[size];
        for(int i=0;i<size;i++)
        {
            A[i] = pt;
            pt.x++;
            pt.y++;
            printf("Punto %d ha ordinata %f\n",i,A[i].y);
        }
    }
}

int main ()
{
    struct Punto pt = {2.4,5.8};
    Prova(pt,10);
}
```

Precedenza e associatività operatore []

<table>
<tr><th>Operatori</th><th>Associatività</th><th>Tipo</th></tr>
<tr><td>[] () ++ (postfisso) -- (postfisso)</td><td>da sinistra a destra</td><td>precedenza più alta</td></tr>
<tr><td>+ - ! ++ (prefisso) -- (prefisso) (tipo)</td><td>da destra a sinistra</td><td>unario</td></tr>
<tr><td>* / %</td><td>da sinistra a destra</td><td>moltiplicativo</td></tr>
<tr><td>+ -</td><td>da sinistra a destra</td><td>additivo</td></tr>
<tr><td>< <= > >=</td><td>da sinistra a destra</td><td>relazionale</td></tr>
<tr><td>== !=</td><td>da sinistra a destra</td><td>di uguaglianza</td></tr>
<tr><td>&&</td><td>da sinistra a destra</td><td>AND logico</td></tr>
<tr><td>||</td><td>da sinistra a destra</td><td>OR logico</td></tr>
<tr><td>?:</td><td>da destra a sinistra</td><td>condizionale</td></tr>
<tr><td>= += -= *= /= %=</td><td>da destra a sinistra</td><td>di assegnazione</td></tr>
<tr><td>,</td><td>da sinistra a destra</td><td>virgola</td></tr>
</table>

Array monodimensionali e funzioni (1/4)

- Le funzioni possono avere come parametri formali variabili di qualsiasi tipo e, dunque, anche variabili array.
- In una funzione, la dichiarazione di un parametro formale di tipo array corrisponde alla dichiarazione di una variabile array ma la lunghezza non viene specificata.

Sintassi parametro formale array monodimensionale:

<tipo elemento> <nome parametro> [ ]

- È possibile utilizzare un'altra notazione per specificare un parametro formale di tipo array (lo vedremo in seguito introducendo i puntatori).
- L'argomento (parametro attuale) in una chiamata di funzione deve essere il **nome di una variabile array** i cui elementi hanno lo stesso tipo degli elementi del parametro formale.

Array monodimensionali e funzioni (2/4)

- Una variabile array è una **variabile puntatore** (il suo valore è l’indirizzo di memoria del primo elemento dell’array).
- Quando in una chiamata di funzione viene passato come argomento una variabile array, viene copiato nel corrispondente parametro formale array l’indirizzo di memoria di partenza dell’array originario.
- Di conseguenza, quando la funzione chiamata modifica nel suo corpo gli elementi del parametro formale array, essa modifica gli effettivi elementi dell’array nelle loro originarie locazioni di memoria.
- In altri termini, la funzione chiamata può modificare i valori degli elementi nell’array specificato come argomento dalla funzione chiamante. Dunque, gli array **vengono passati per riferimento**.

Array monodimensionali e funzioni (3/4)

- Il passaggio degli array per riferimento ha senso per ragioni di prestazioni. Se gli array fossero passati per valore, verrebbe passata **una copia** di ogni elemento. Per array grandi, passati frequentemente, ciò comporterebbe un *overhead* considerevole sia in termini di tempo di computazione che di quantità di memoria.
- BUONA NORMA: in funzioni che prendono in input parametri array monodimensionali, utilizzare un altro parametro intero per specificare la lunghezza dell'array od il numero di elementi dell'array da leggere o modificare.

Array monodimensionali e funzioni (4/4)

Esempio:

```
#define SIZE 5

void ModificaArray(int b[],int size)
{
    if(size>0)
    {
        for(int j =0;j<size;j++)
            b[j] *= 2;
    }
}

int main ()
{
    int a[SIZE] ={0,1,2,3,4};

    //stampa l'array originario
    for(int i=0; i<SIZE;i++)
        printf("Elemento %d di array originario: %d\n",i,a[i]);

    ModificaArray(a,SIZE);

    //stampa l'array modificato
    for(int i=0; i<SIZE;i++)
        printf("Elemento %d di array modificato: %d\n",i,a[i]);

}
```

Output:

```
Elemento 0 di array originario: 0
Elemento 1 di array originario: 1
Elemento 2 di array originario: 2
Elemento 3 di array originario: 3
Elemento 4 di array originario: 4
Elemento 0 di array modificato: 0
Elemento 1 di array modificato: 2
Elemento 2 di array modificato: 4
Elemento 3 di array modificato: 6
Elemento 4 di array modificato: 8
```

Esempio utilizzo array monodimensionali

Calcolo somma degli elementi di un array di tipo numerico.

```
int CalcolaSomma(int b[], int size)
{
	int res = 0;
	if(size>0)
	{
		for(int j =0;j<size;j++)
			res += b[j];
	}
	return res;
}
```