Titolo unità didattica: Array e insiemi [08]

Titolo modulo : Tipi enumerativi e tipi derivati in C [04-C]

Proprietà dei tipi enumerativi in C e generazione di tipi derivati

Argomenti trattati:

- ✓ istruzione **`typedef`**
- ✓ comando **`#define`**
- ✓ tipi **`enum`**
- ✓ esempi di function C con parametri di tipo enumerativo e derivato

Prerequisiti richiesti: AP-03-03-C, AP-05-04-C

tipi derivati da tipi primitivi

in C si può dare un **nome** (*identificatore*) a un tipo esistente

```
typedef <tipo_esistente> <nuovo_tipo>;
```

```
typedef int Integer;
typedef char Character;
```

il **nome** (*identificatore*) è utilizzabile come un **nuovo tipo**

il nuovo tipo può essere utilizzato per dichiarare variabili o funzioni

```
Integer i,j;
Character c, parola[5];
Character int2char(Integer);/*prototipo*/
```

tipi derivati da tipi primitivi

in C si può dare un **nome** (*identificatore*) a un tipo esistente

```
typedef <tipo_esistente> <nuovo_tipo>;
```

```
typedef int Integer;
typedef char Character;
```

il **nome** (*identificatore*) è utilizzabile come un **nuovo tipo**

il nuovo tipo può essere utilizzato per dichiarare variabili o funzioni

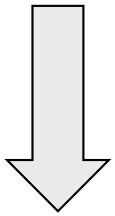
sono equivalenti

```
Integer i,j;
Character c, parola[5];
Character int2char(Integer);
```

```
int i,j;
char c, parola[5];
char int2char(Integer);
```

tipi derivati da tipi primitivi

in C si può dare un **nome** (*identificatore*) a un tipo esistente



migliorare la *leggibilità* del programma
il nome di un nuovo tipo può riflettere l'uso che si intende fare di quel tipo nel programma

```
typedef int Euro;
typedef int Ore_lavoro ;
Euro stipendio,costo;
Euro reddito(Ore_lavoro);/*prototipo*/
```

comando (al precompilatore) #define

```
#define <identificatore> <valore>
```

il precompilatore sostituisce,
nel testo del programma sorgente, tutte le occorrenze di
<identificatore> con **<valore>**

```
#define PI_GRECO 3.1415926F
#define TRUE 1
#define FALSE 0
```

```
area_cerchio = PI_GRECO*raggio*raggio;
…
if (a == b)
   uguale = TRUE;
 else
   uguale = FALSE;
```

comando (al precompilatore) **#define**

```
#define <identificatore> <valore>
```

il precompilatore sostituisce,
nel testo del programma sorgente, tutte le occorrenze di
<identificatore> con **<valore>**

```
#define PI_GRECO 3.1415926F
#define TRUE 1
#define FALSE 0
```

```
area_cerchio = 3.1415926F*raggio*raggio;
…
if (a == b)
   uguale = 1;
 else
   uguale = 0;
```

comando (al precompilatore) **#define**

```
#define <identificatore> <valore>
```

il precompilatore sostituisce,
nel testo del programma sorgente, tutte le occorrenze di
<identificatore> con **<valore>**

```
#define N 100
#define N_RIGHE 1024
#define N_COLONNE 1024
```

```
float a[N];
int foto_bn[N_RIGHE][N_COLONNE];
for (i=0;i<N;i++)…
```

```
float a[100];
int foto_bn[1024][1024];
for (i=0;i<100;i++)…
```

comando (al precompilatore) **#define**

```
#define <nome_macro(x)> <istruzioni_macro>
```

il precompilatore sostituisce,
nel testo del programma sorgente, tutte le occorrenze di
<nome_macro> con **<macro>** e
trattando l'identificatore **x** come parametro

```
#define QUADRATO(x)  ((x)*(x))
…
a = QUADRATO(b);
c = QUADRATO(a+b);
```

```
a = ((b)*(b));
c = ((a+b)*(a+b));
```

tipi derivati da tipi primitivi

in C si possono creare *gerarchie* di tipi derivati

```
#define N 12
typedef double Scalare;
typedef Scalare Vettore[N];
typedef Scalare Matrice_quadrata[N][N];
```

sono stati creati i tipi
Scalare **Vettore** **Matrice_quadrata**

```
Scalare velocita_luce = 299792.458;
Vettore inflazione_mese;
Matrice_quadrata M,A,B;
```

```
double velocita_luce = 299792.458;
double inflazione_mese[12];
double M[12][12],A[12][12],B[12][12];
```

tipi derivati da tipi primitivi

in C si possono creare *gerarchie* di tipi derivati

```
#define N 12
typedef double Scalare;
typedef Scalare Vettore[N];
typedef Scalare Matrice_quadrata[N][N];
```

```
void somma(Vettore x,Vettore y,Vettore z)
{
 int i;
 for (i=0;i<N;i++)
      z[i] = x[i] + y[i];
}
```

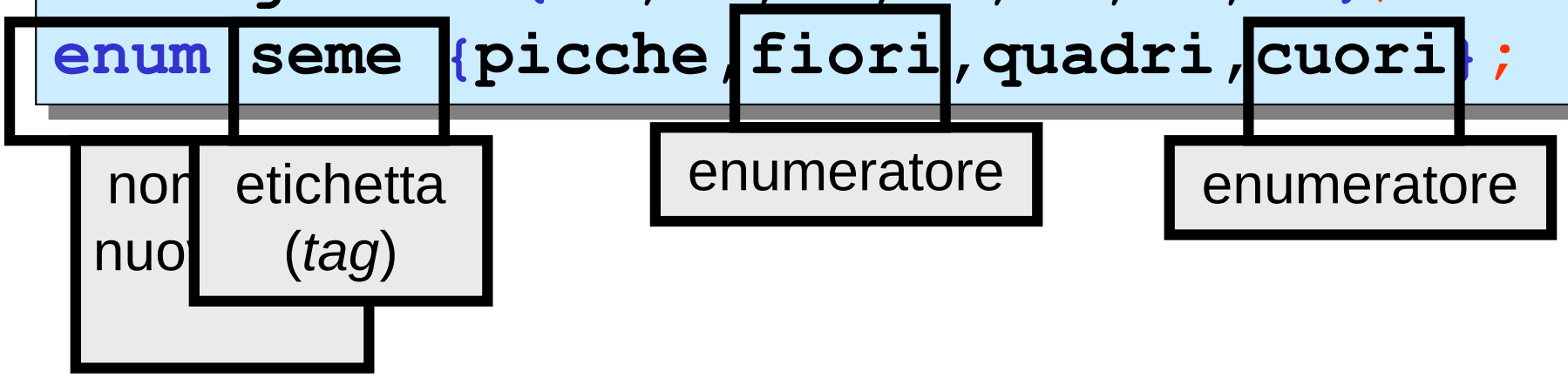
Attenzione: la function `somma` opera solo su vettori di **size fissato**

tipi enumerativi

in C si può creare un nuovo tipo enumerando tutti i suoi oggetti (tipo *enumerativo*)

```
enum <etichetta> {<elenco_oggetti>};
```

```
enum giorno {Lu,Ma,Me,Gi,Ve,Sa,Do};
enum seme {picche,fiori,quadri,cuori};
```



un enumeratore è una costante (simbolica) di tipo `int`

tipi enumerativi

in C si può creare un nuovo tipo enumerando tutti i suoi oggetti (tipo *enumerativo*)

```
enum <etichetta> {<elenco_oggetti>};
```

```
enum giorno {Lu,Ma,Me,Gi,Ve,Sa,Do};
enum seme {picche,fiori,quadri,cuori};
```

il nuovo tipo (`enum <etichetta>`) può essere utilizzato per dichiarare variabili o funzioni

```
enum giorno festa,riunione;
enum seme carta[40];
enum giorno chegiornoe(int);/*prototipo*/
```

tipi enumerativi

le seguenti dichiarazioni sono ammissibili

`festa` è una variabile di tipo `enum giorno`

```
enum giorno {Lu,Ma,Me,Gi,Ve,Sa,Do}festa;
```

```
enum {Lu,Ma,Me,Gi,Ve,Sa,Do} festa;
```

l'etichetta non è necessaria

`Giorno` è un nuovo tipo

```
typedef enum {Lu,Ma,Me,Gi,Ve,Sa,Do} Giorno;
```

```
typedef enum giorno Giorno;
```

un **tipo enumerativo** è equivalente al tipo `int` nel senso che:

- ✓ gli enumeratori (per es. `Lu`, `Ma`, `quadri`,..) sono costanti (simboliche) di tipo `int`
- ✓ il valore del primo enumeratore è `0`, del secondo è `1`, e così via

```
enum giorno {Lu,Ma,Me,Gi,Ve,Sa,Do};
```

è equivalente a

```
const int Lu = 0;
const int Ma = 1;
const int Me = 2;
const int Gi = 3;
const int Ve = 4;
const int Sa = 5;
const int Do = 6;
```

```
#define Lu 0
#define Ma 1
#define Me 2
#define Gi 3
#define Ve 4
#define Sa 5
#define Do 6
```

un **tipo enumerativo** è equivalente al tipo `int` nel senso che:

- ✓ gli enumeratori (per es. `Lu`, `Ma`, `quadri`,..) sono costanti (simboliche) di tipo `int`
- ✓ il valore del primo enumeratore è `0`, del secondo è `1`, e così via

```
enum giorno {Lu=1,Ma,Me,Gi,Ve,Sa,Do};
```

il valore del primo enumeratore è `1`, del secondo è `2`, e così via

```
enum giorno festa,riunione;
festa = Do;
riunione = festa;
riunione = Lu ;
if(riunione == Lu)
   printf(" %d \n",riunione);
```

```
1
```

un **tipo enumerativo** è equivalente al tipo `int` nel senso che:

- ✓ gli enumeratori (per es. `Lu`, `Ma`, `quadri`,..) sono costanti (simboliche) di tipo `int`
- ✓ il valore del primo enumeratore è `0`, del secondo è `1`, e così via

```
enum giorno {Lu,Ma,Me,Gi,Ve,Sa,Do};
```

- un enumeratore occupa lo stesso **spazio di memoria** di un dato del tipo `int`
- agli enumeratori si possono applicare gli stessi **operatori** del tipo `int`
- gli enumeratori possono essere usati in **espressioni di confronto**
- se si definiscono più tipi `enum` **non** si possono **riutilizzare gli stessi enumeratori**
- **non** è possibile assegnare un valore a un enumeratore con una istruzione di assegnazione (~~`Lu=1;`~~)

```
enum cabala {cane=1,gatto=3,topo=7,spada=18,foglie=78,
   … paura=90};
```

agli enumeratori possono essere assegnati valori numerici anche **indicando direttamente il simbolo ASCII**

```
enum simbolo{nome,numero,fine,piu ='+',
     meno ='-',per ='*',div ='/'};
```

è equivalente a

```
const int nome = 0;
const int numero = 1;
const int fine = 2;
const int piu = 43;  /* rappr. ASCII di ='+'  */
const int meno = 45; /* rappr. ASCII di ='-'  */
const int per = 42;  /* rappr. ASCII di ='*'  */
const int div = 47;  /* rappr. ASCII di ='/'  */
```

realizzazione del tipo derivato **Logical**

```
typedef enum {false,true} Logical;
Logical trovato;
…
trovato = true; /* o anche trovato = false;*/
…
if(trovato)
   printf("trovato e'vero %d",trovato);
 else
   printf("trovato e'falso %d",trovato);
```

è importante che il valore dell'enumeratore
false sia **0** e il valore di **true** sia **1**,
al fine di mantenere la compatibilità con le operazioni logiche del C

```
trovato e' vero 1
```

Esempio: function che restituisce il giorno successivo di un giorno, passato come parametro di input, e **main** di chiamata

```
/* determinazione del giorno successivo
     (Lu,Ma,Me,...) di un giorno dato   */
#include <stdio.h>
enum giorno {Lu,Ma,Me,Gi,Ve,Sa,Do};
typedef enum giorno Giorno;
Giorno giorno_dopo(Giorno);
void main()
{
  Giorno oggi,domani;
  oggi = Me;
  domani = giorno_dopo(oggi);
  printf(" se oggi e' %d allora domani e'
          %d\n",oggi,domani);
}
```

```
Giorno giorno_dopo(Giorno x)
{
  Giorno y;
  switch (x){
    case Lu: y = Ma; break;
    case Ma: y = Me; break;
    case Me: y = Gi; break;
    case Gi: y = Ve; break;
    case Ve: y = Sa; break;
    case Sa: y = Do; break;
    case Do: y = Lu; break;
  }
 return y;
}
```

```
se oggi e' 2 allora domani e' 3
```

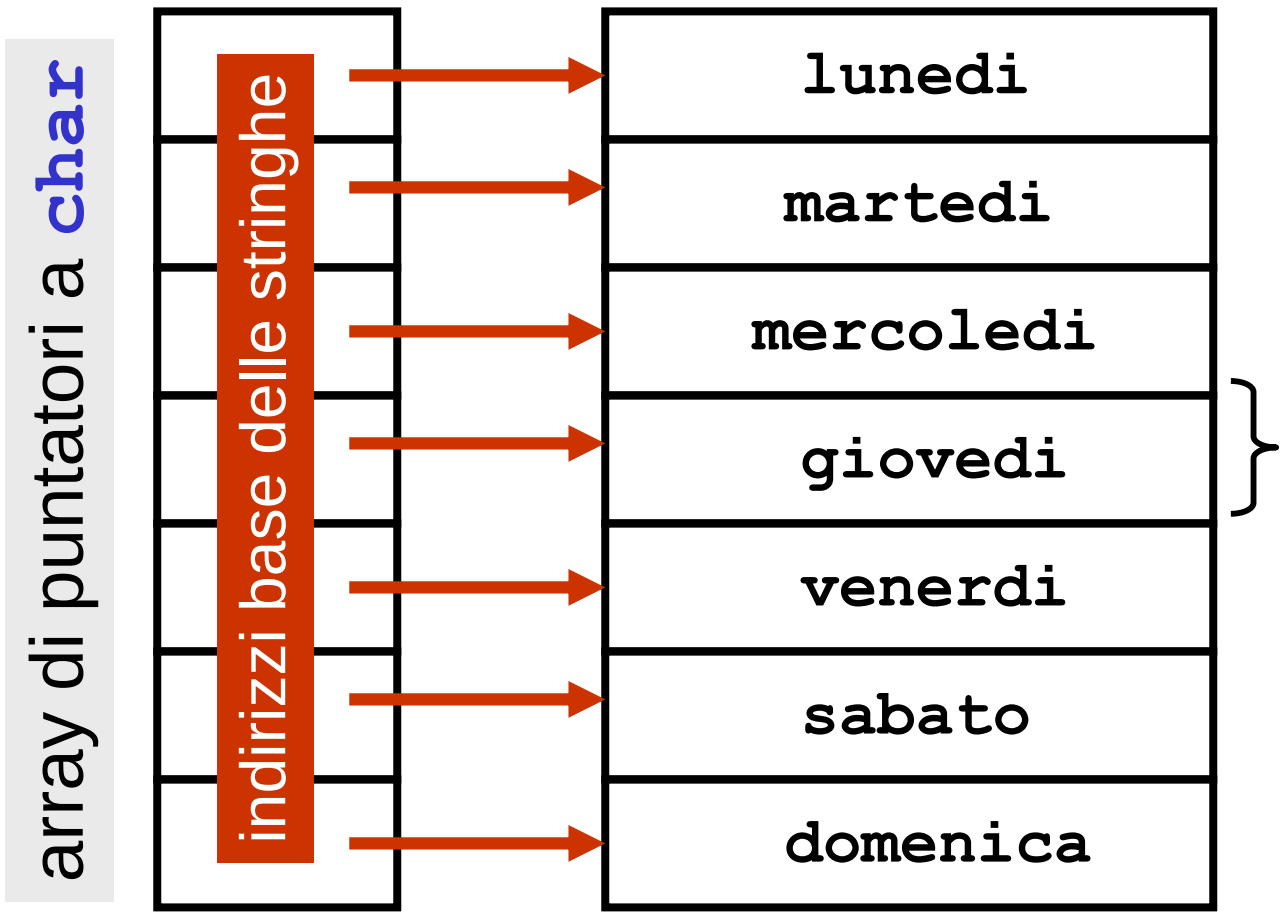
Esempio: function che restituisce il giorno successivo di un giorno passato come parametro di input e **main**

```
/* determinazione del giorno successivo
     (Lu,Ma,Me,...)                          */
#include <stdio.h>
enum giorno {Lu,Ma,Me,Gi,Ve,Sa,Do};
typedef enum giorno Giorno;
Giorno giorno_dopo(Giorno);
void main()
{
  Giorno oggi,domani;
  char *giorni_settimana[]={"lunedi","martedi",
          "mercoledi","giovedi","venerdi",
          "sabato","domenica"};
```

array di puntatori a **char**

array di puntatori a `char`

array di puntatori a stringhe costanti



`giorni_settimana`

`stringhe`

```
 oggi = Me;
 domani = giorno_dopo(oggi);
 printf(" se oggi e' %d allora domani e'
         %d\n",oggi,domani);
 printf(" se oggi e' %s allora ",
         giorni_settimana[(int)oggi]);
 printf(" domani e' %s\n",
         giorni_settimana[(int)domani]);
}
```

puntatore

puntatore

cast: valore di **oggi** e di **domani** come interi

```
se oggi e' 2 allora domani e' 3
se oggi e' mercoledi allora domani e' giovedi
```

versione alternativa della function `giorno_dopo`

```
Giorno giorno_dopo(Giorno x)
{
 return ((Giorno)(((int)x +1)%7));
}
```

cast: il valore dell'espressione è trasformato nel tipo `Giorno`

cast: valore di `x` come intero

modulo: resto della divisione intera