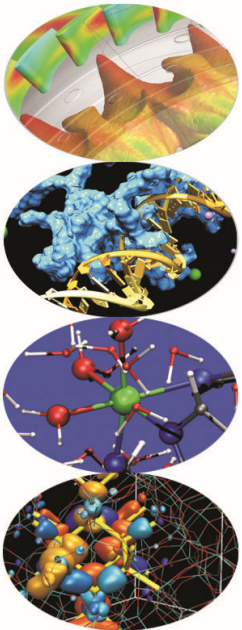
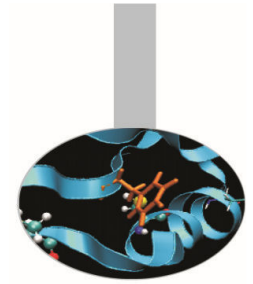


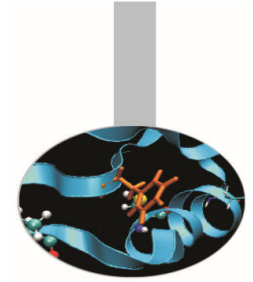
Dati Strutturati



Indice



- **L'istruzione enum**
- **L'istruzione typedef**
- **Le struct**
- **Le union**



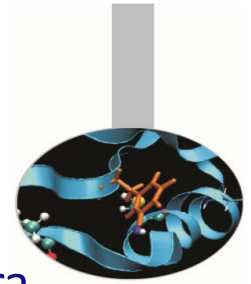
Tipi di dato derivato

A partire dai tipi di dato base, vengono creati i cosiddetti tipi derivati. Un array è un esempio di tipo di dato derivato in cui tutti gli elementi sono dello stesso tipo. Non sempre questo tipo di dato è sufficiente a rappresentare correttamente la variabile del nostro problema.

Gli altri tipi derivati sono:

- Enumerazioni
- Istruzione typedef
- Struct
- Union

Enumerazioni



Il tipo di dato enumerazione (enum) viene utilizzato per gestire in maniera raffinata un insieme di valori interi specificati dall'utente.

```
enum nome_enumerazione{lista_elementi};
```

```
enum{bianco, rosso, verde, blue=10, nero,  
     arancio=blue+8};
```

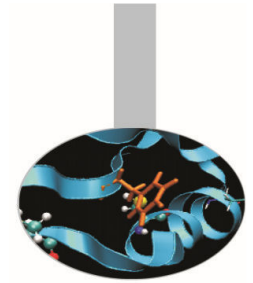
È del tutto equivalente a:

```
const int bianco=0, rosso=1, verde=2, blue=10,  
        nero=11, arancio=18;
```

Per default il primo elemento ha valore zero e tutti gli elementi che lo seguono hanno valori incrementali rispetto al precedente

Una volta che l'enumerazione è stata creata con un nome, questa costituisce un nuovo tipo di dato

Enumerazioni



La conversione da enum ad intero è implicita, il viceversa no

```
enum condizione {alto,medio,basso};
```

```
condizione x, y;
```

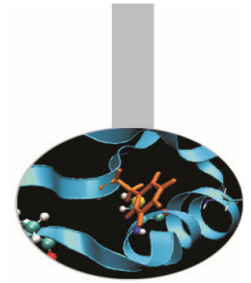
```
x = alto;    // ok, variabile di tipo condizione  
             //  inizializzata
```

```
y = 1;      // errore: non esiste una conversione da  
           //  intero a condizione
```

```
y = condizione(1); //ok, conversione esplicita di tipo,  
                  //  da int a condizione
```

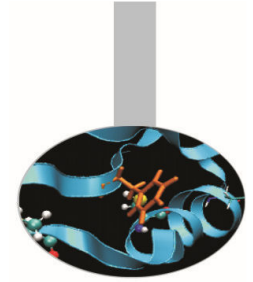
```
int i = basso; //ok, accettata la conversione da enum  
              //  a int
```

Enumerazioni



```
/*l'uso tipico del costrutto enum è insieme al costrutto switch*/
enum cond_cont {Dirichlet=1, Neumann, Robin};
cond_cont a;
a = Robin; //equivalente a scrivere a=cond_cont(3)

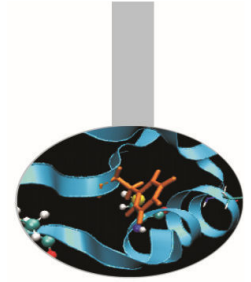
switch(a) {
case Dirichlet:
    /* Condizioni al contorno Dirichlet */
    break;
case Neumann:
    /* Condizioni al contorno Neumann */
    break;
case Robin:
    /* Condizioni al contorno Robin */
    break;
Default:
    fprintf(stderr, "Condizione al contorno non definita\n");
}
```



Istruzione typedef

- Il C è un linguaggio fortemente tipizzato, per cui ogni variabile è associata ad un tipo di dato.
- Tramite l'istruzione *typedef* è però possibile definire un nuovo nome associato a un tipo di dato, anche definito dal programmatore.
- L'effetto che si ottiene è un codice più facilmente leggibile.
- Di fatto non si sta però definendo nessun nuovo tipo di dato.
- Solo le classi del C++ permettono di definire un nuovo tipo di dato, che abbia gli stessi privilegi sintattici rispetto a quelli forniti dal linguaggio.

Istruzione typedef



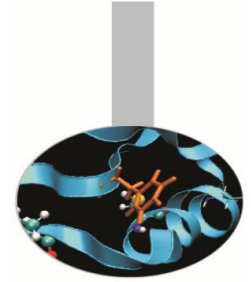
```
/* esempi */
```

```
int fagioli;                typedef int Legumi;  
fagioli=18;                Legumi fagioli=18;  
  
/*queste due versioni sono equivalenti per il  
  compilatore*/
```

```
typedef enum{FALSE=0,TRUE} Boleani;  
Boleani flag=TRUE;
```

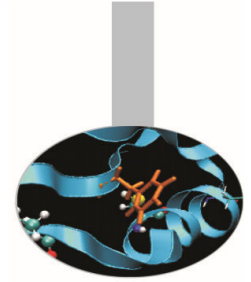
```
typedef struct{int matricola, int data_di_nascita;  
  char *nome; char *cognome;} Anagrafe;  
Anagrafe DipendentiSPA[1000], ImpiegatiXXX[20];  
  
/*il tipico esempio di utilizzo di typedef è per  
  nominare dati strutturati*/
```


C-struct



- Il tipo di dato strutturato eterogeneo per eccellenza in C è la *struct*
- Per accedere ai dati viene usato l'operatore (.).
- In C non si possono dichiarare funzioni all'interno di *struct*.

C-struct

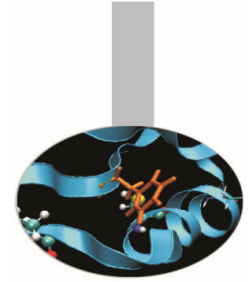


```
/* esempi */
```

```
struct mystruct { //nome della struct  
    int mydata1;    //membro1 della struttura  
    double mydata2; //membro2 della struttura  
    char mydata3;  //membro3 della struttura  
}; /*chiusura blocco di istruzioni struct */
```

```
struct mystruct myvar1, myvar2;
```

C-struct

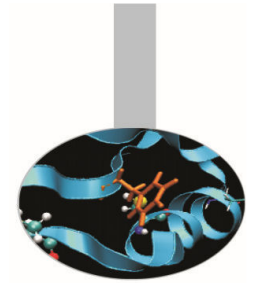


```
/* esempi */
```

```
/* posso definire le variabili anche contestualmente  
   alla definizione della struct */
```

```
struct mystruct {  
    int mydata1;  
    double mydata2;  
    char mydata3;  
} myvar1, myvar2;
```

C-struct

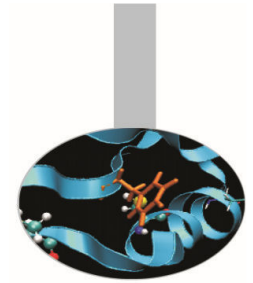


Usando *struct* si possono gestire facilmente gruppi di variabili logicamente interconnesse tra loro.

Raggruppando le variabili in una *struct* risulta più agevole modificare/gestire il codice che le manipola.

```
/* Inizializzazione: */  
struct mystruct myvar1={3,88.6,'p'};  
struct mystruct myvar2={5,55.3,'r'};  
myvar1.mydata1=1;    /* accesso ai singoli  
membri */  
myvar1.mydata2=90.0;  
myvar1.mydata3= 'X';  
myvar2=myvar1; /* copia membro per membro */
```

C-struct

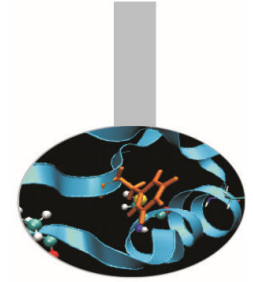


```
struct mystruct{ /*strutture contenti array*/  
    int mydata1[2];  
    double mydata2[5];  
    char mydata3;  
}; mystruct myvar1={{3,5},{88.6,43.7,77.9},'p'};
```

```
myvar1.mydata1[0]=1;  
myvar1.mydata1[1]=2;  
myvar1.mydata2[3]=44.70;  
myvar1.mydata2[4]=90.0;  
myvar1.mydata3= 'X';
```

Per l'accesso ai membri di fatto si usa l'operatore (.) in congiunzione con l'operatore ([])

C-struct



```
struct mystruct{ /* array di struct */  
    int mydata1[2];  
    double mydata2[5];  
    char mydata3; } many[3];
```

```
many[0].mydata1[1] = 33;
```

```
/* nel primo elemento dell'array many si modifica il  
    secondo valore del membro mydata1 */
```

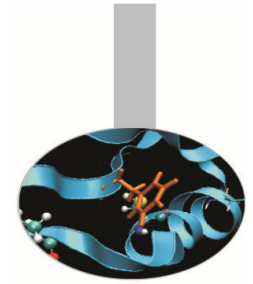
```
many[1].mydata2[3] = 77,5;
```

```
/* nel secondo elemento dell'array many si modifica  
    il quarto valore del membro mydata2 */
```

```
many[2].mydata3 = 'g';
```

```
/* nel terzo elemento dell'array many si modifica il  
    valore del membro mydata3 */
```

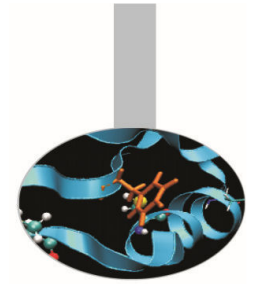
C-struct



```
struct inside { /* struct con membri struct */
    double mydatainside1;
    char mydatainside2;
};
struct mystruct{
    int mydata1;
    struct inside mydata2;
} myvar1;
```

```
myvar1.mydata1 = 3;
myvar1.mydata2.mydatainside1 = 65.8;
myvar1.mydata2.mydatainside2 = 'u';
```

C-struct

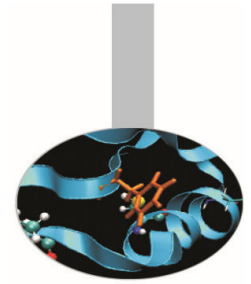


```
typedef struct { /* uso di typedef */
    int data1;
    int data2;
} MY_S;
```

In questo modo si è definito un typedef per MY_S ma attenzione che questa non è una variabile ma un nome alternativo per dichiarare una variabile struttura

```
int main() {
    MY_S my_var_s; // ok, usiamo il typedef
    my_var_s.data1=6; //ok usiamo la variabile
    my_var_s.data2=52;
    MY_S.data1 = 5; /* errore MY_S è il nome del
        typedef, non di una variabile */
}
```


C-struct

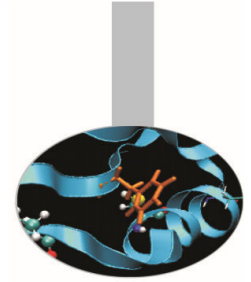


Esempio coda-semplice

Le struct permettono di realizzare strutture dati flessibili quali code, pile, liste. La base per realizzare queste costruzioni è definire una struct contenente un puntatore a sé stessa:

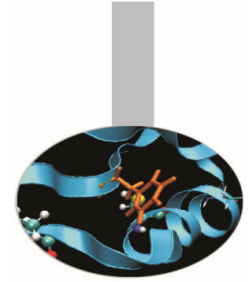
```
struct Anello {  
    int Val;  
    struct Anello *Prossimo;  
};
```

C-struct



```
struct Anello { int Val; struct Anello *Prossimo; };
int main () { /* file e-coda.c */
    struct Anello *coda, *corr, *prox;
    . . .
    for ( i = 0; i < 10; i++ ) {
        prox = malloc(sizeof(struct Anello));
        prox->Val = i;      prox->Prossimo = NULL;
        if ( i == 0 ) { /* Primo anello */
            coda = prox; }
        else { corr->Prossimo = prox;
        }
        corr = prox;
    } return(0); }
```

C-struct



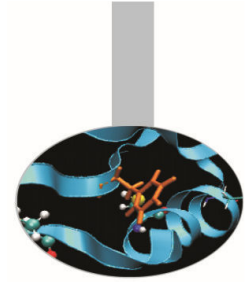
Esempio coda-semplice-2

Volendo facilitare l'uso della coda, si può riscrivere l'esempio raggruppando i puntatori in una struttura e modularizzare le operazioni in apposite funzioni:

```
struct Anello {
    int Val;    struct Anello *Prossimo;
};

struct Coda{
    int anelli;
    struct Anello *primo, *corr, *prox;
};
```

C-struct

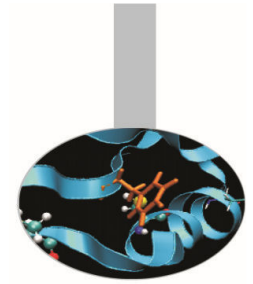


```
struct Anello {
    int Val; struct Anello *Prossimo; };

struct Coda{
    int anelli;
    struct Anello *primo, *corr, *prox; };

void InizializzaCoda(struct Coda *this) {
    (*this).anelli=0;
};
```

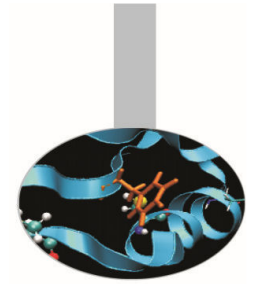
C-struct



```
int AggiornaCoda(struct Coda *this, int num) {
    this->prox = malloc(sizeof(struct Anello));

    (*this->prox).Val = num;
    this->prox->Prossimo = NULL;
    if ( (*this).anelli == 0 ) {
        (*this).anelli = 1;
        this->primo = this->prox;
    } else {
        this->corr->Prossimo = this->prox;
        (*this).anelli = (*this).anelli + 1;
    }
    this->corr = this->prox;
    return 0; }
```

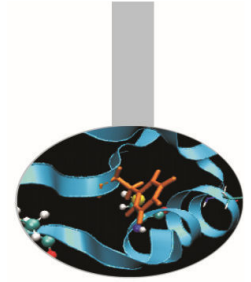
C-struct



```
int AnelliCoda(struct Coda this) {  
    return this.anelli;  
}
```

```
void StampaCoda(struct Coda this) {  
    this.prox = this.primo;  
    do {  
        printf("%d\n",this.prox->Val);  
        if ( this.prox->Prossimo == NULL ) break;  
        this.prox = this.prox->Prossimo;  
    } while (1 ==1);  
    printf("\n");  
}
```

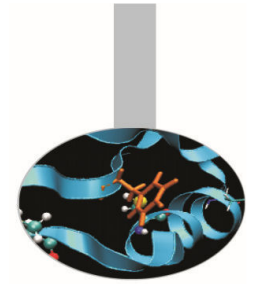
C-struct



```
int main () {
    struct Coda numeri;
    int i;
    InizializzaCoda (&numeri);
    for ( i = 0; i < 10; i++ ) {
        AggiornaCoda (&numeri, i);
    }
    printf("Sono stati inseriti %d numeri:\n",
        AnelliCoda (numeri));
    StampaCoda (numeri);

    return (0);
}
```

Union



Le *union* sono particolari *struct* atte a salvare spazio in memoria.

Solo un membro di una union può esistere in un determinato istante di tempo.

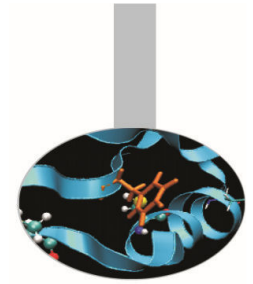
Lo spazio massimo occupato in memoria da una union è pari allo spazio massimo occupato dal suo membro più grande.

Generalmente vengono usate all'interno di *struct* per definire dati che possono esistere solo alternativamente (vedi esempio).

```
union poli_tipo {           // union con tre membri
    int j;
    char a;
    double b;           // massimo spazio occupato
};

poli_tipo mix;           // mix è una variabile poli_tipo
```

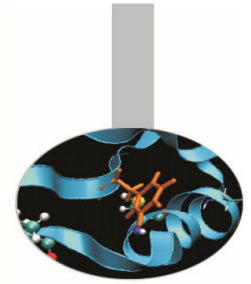

Esempio: poli-tipo



```
#include <stdio.h> /* file e-poli_tipo.c */
typedef union {
    int j;  char a;  double b;
} poli_tipo ;
int main() { /* memoria allocata: sizeof(double) */
    poli_tipo mix;
    printf("solo il membro j viene utilizzato");
    mix.j = 9;
    printf(" membro j = %d;\t membro a = %c;\t membro
           b = %lf\n",mix.j,mix.a,mix.b);

    printf("solo il membro a viene utilizzato");
    mix.a = 'P';
    printf(" membro j = %d;\t membro a = %c;\t membro
           b = %lf\n",mix.j,mix.a,mix.b);
```

Esempio: poli-tipo



```
printf("solo il membro b viene utilizzato");  
mix.b = 56.9;  
printf(" membro j = %d;\t membro a = %c;\t membro  
      b = %lf\n",mix.j,mix.a,mix.b);  
  
return 0;}
```

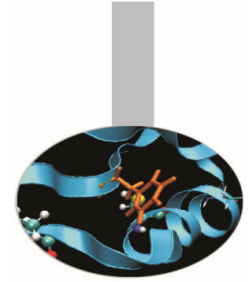
Output:

```
solo il membro j viene utilizzato membro j = 9; membro a =  
membro b = 0.000000
```

```
solo il membro a viene utilizzato membro j = 80; membro a = P;  
membro b = 0.000000
```

```
solo il membro b viene utilizzato membro j = 858993459; membro a  
= 3; membro b = 56.900000
```

Commenti

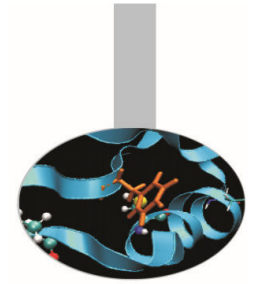


L'esempio mostra come essendo definibili ogni volta solo uno dei tre dati, il valore degli altri due è imprevedibile.

Il compilatore non pone alcun vincolo su questo.

E' compito del programmatore utilizzare correttamente i dati che sono definiti in ogni momento.

Esempio: poli-tipo array

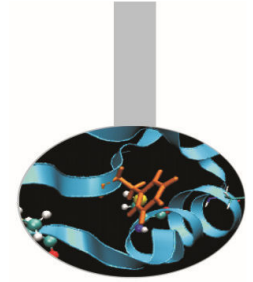


```
#include <stdio.h>  /* file e-poli_array.c */
typedef union {
    int j;  char a;  double b;
} poli_tipo ;

int main() {
    poli_tipo mix[10];

    mix[0].j = 0;           mix[1].a = 'a';
    mix[2].b = 0.123;      mix[3].j = 123;
    mix[4].a = 'Z';        mix[5].b = 9.12345678;
    mix[6].j = 18;         mix[7].a = 'q';
    mix[8].b = 0.0012;     mix[9].j = 13;
```

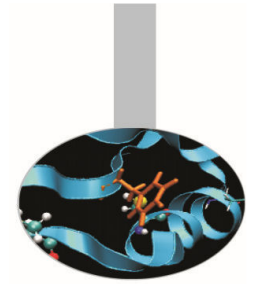
Esempio : poli-tipo array



```
for ( i = 0; i < 10; i++ ) {  
    if ( i%3 == 0 ) printf("mix[%d] =  
%d\n",i,mix[i].j);  
    if ( i%3 == 1 ) printf("mix[%d] =  
%c\n",i,mix[i].a);  
    if ( i%3 == 2 ) printf("mix[%d] =  
%lf\n",i,mix[i].b);  
}  
return 0;  
}
```

```
Output:  
mix[0] = 0  
mix[1] = a  
mix[2] = 0.123000  
mix[3] = 123  
mix[4] = Z  
mix[5] = 9.123457
```

Esempio union in struct

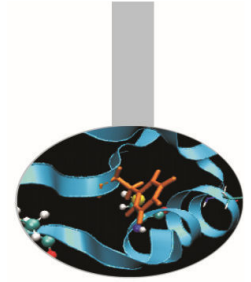


Si supponga di avere una struttura figura2d (quadrato o triangolo) ma che ovviamente non può essere contemporaneamente un triangolo e un quadrato

```
struct figura2d {
    char[20] nome;
    bool tipo; /* 0 se triangolo, 1 se quadrato (etichetta di tipo)*/
    union {
        triangolo tria; // triangolo è a sua volta una struct
        quadrato quad; // quadrato è a sua volta una struct
    }
}

int main() {
    figura2d fig1;
    fig1.nome="figural";
    fig1.tipo=0;
    fig1.tria.base= 12.9;
    fig1.tria.altezza=5.5; }
```

Commenti



Dal momento che una *figura 2d* non può essere contemporaneamente un triangolo o un rettangolo, con l'uso di *union* si evidenzia questo aspetto

Si noti comunque come tutto il peso dell'implementazione corretta sia a carico del programmatore