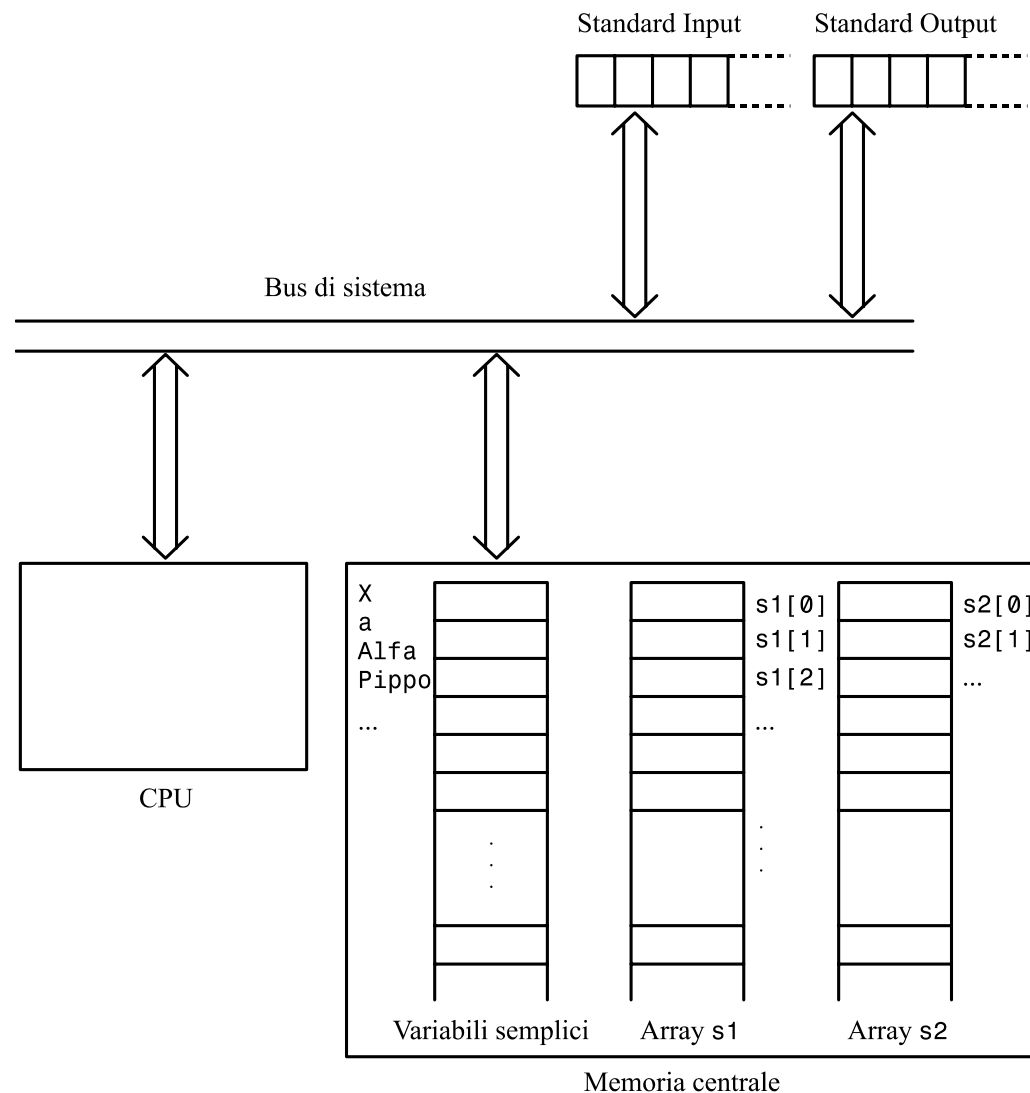


## Le variabili strutturate

Un primo arricchimento della macchina astratta C



## Gli array

Il **vettore** o *array monodimensionale* è costituito da *elementi* in cui è possibile memorizzare valori di tipo omogeneo.

Ogni elemento è individuato da un numero progressivo, detto *indice*.

L'indice può assumere valori interi da zero al numero totale di elementi meno 1.

Il numero complessivo degli elementi del vettore viene detto *lunghezza*.

Un **array** viene identificato come qualsiasi altra variabile.

*esempi:*

```
scanf( "%d" , &s[2] );  
a[3] = s[1] + x;  
if (a[4] > s[1] + 3) s[2] = a[2] + a[1];  
x = a[i];  
a[i] = a[i+1];  
a[i*x] = s[a[j+1]-3]*(y - a[y]);
```

N.B.

In C il primo elemento di ogni array è sempre lo 0-esimo

## I tipi strutturati: il costruttore array

## [ ]

Definizione del nuovo tipo **vettore**:

```
typedef    int    vettore[100];    ⇒    costruisce tipo
```

Dichiarazione di variabili di tipo **vettore**:

```
vettore    alfa,beta;
```

*o, in alternativa,*

```
int        alfa[100], beta[100];    ⇒    costruisce direttamente variabili
```

La dichiarazione:

```
int        lista[20]; va perciò interpretata come un'abbreviazione per:
```

```
typedef    int        vettoregenerico[20];  
vettoregenerico    lista;
```

***Il valore dell'indice di un array di N elementi varia da 0 ad N-1.***

Indicare valori dell'indice al di fuori di tale intervallo produce un errore di indirizzamento.

## Osservazioni

Quando conviene esplicitare il nome del tipo?

La dichiarazione mediante nuovo tipo:

```
typedef    double    VettoreDiReali[20];  
VettoreDiReali    v1, v2, v3;
```

Più semplice e altrettanto chiara è quella abbreviata:

```
double    v1[20], v2[20], v3[20];
```

La dichiarazione mediante nuovi tipi:

```
typedef    double    PioggeMensili[12];  
typedef    double    IndiciBorsa[12];  
PioggeMensili    Piogge01, Piogge02, Piogge03;  
IndiciBorsa    Indici01, Indici02, Indici03;
```

è in questo caso preferibile a:

```
double    Piogge01[12], Piogge02[12], Piogge03[12],  
Indici01[12], Indici02[12], Indici03[12];
```

## Osservazioni

### Matrici

E' possibile definire un *array di array* (una *matrice*):

```
typedef      int      Vettore[20];  
typedef      Vettore  MatriceIntera20Per20[20];  
MatriceIntera20Per20  matrice1;
```

Oppure, più brevemente:

```
typedef      int      MatriceIntera20Per20[20][20];  
MatriceIntera20Per20  matrice1;
```

Anche una matrice può essere definita in modo *abbreviato*:

```
int      matrice1[20][20];
```

E' possibile definire array di array di array...:

```
int      matriceTridimensionale1[10][20][30];
```

Per accedere agli elementi di `matriceTridimensionale1`:

```
matriceTridimensionale1[2][8][15]
```

E' possibile definire array di ...:

```
colore      ListaColori[10];
```

## Osservazioni

### Dimensione degli array

*Un array ha dimensioni fisse* -> minor flessibilità del linguaggio

```
typedef      char      String[30];  
String      Nome, Cognome;
```

*La rigida e preliminare definizione delle dimensioni dell'array può comportare **spreco di memoria** (se si utilizzano solo alcuni degli elementi dell'array) oppure **errore di indirizzamento** (se non si controlla che il valore dell'indice rimanga nell'intervallo fissato).*

Parole corte provocano spreco di memoria (fisica)

Parole lunghe: dovremmo anche prevedere istruzioni del tipo:

```
if (LunghezzaParola == 30)  
    printf("Parola troppo lunga");
```

Perché?

*Principio dell'allocazione statica della memoria.*

***Gli estremi di variabilità non possono cambiare durante l'esecuzione del programma.***

L'allocazione della memoria richiesta dal programma - e quindi anche dalle variabili - viene infatti eseguita, salvo casi particolari, prima dell'inizio dell'esecuzione (allocazione statica).

## I primi, semplici, programmi con array in C

```
/* Programma InvertiSequenza di max 99 numeri */
#include <stdio.h>
main()
{
    int indice, x, sequenza[99];
    indice = 0;
    scanf("%d", &x);
    while (indice < 99 && x != '%')
    {
        sequenza[indice] = x;
        indice = indice + 1;
        if (indice < 99) scanf("%d", &x);
    }
    while (indice > 0 && indice <= 99)
    {
        indice = indice - 1;
        printf("indice %d valore %d\n", indice, sequenza[indice]);
    }
}
```

E se invece di 99 la sequenza fosse lunga max 1000 numeri?

Un parziale rimedio alle dimensioni fisse di un array: la “**parametrizzazione**”.

Questa consiste nell'uso di costanti per definire le dimensioni di array, eventualmente ricorrendo alla *direttiva* **# define**.

```
/* Programma InvertiSequenza di max 1000 numeri */
#include <stdio.h>
#define lung 1000
main()
{
    int indice, x, sequenza[lung];
    indice = 0;
    scanf("%d", &x);
    while (indice < lung && x != '%')
    {
        sequenza[indice] = x;
        indice = indice + 1;
        if (indice < 99) scanf("%d", &x);
    }
    while (indice > 0 && indice <= lung)
    {
        indice = indice - 1;
        printf("indice %d valore %d\n", indice, sequenza[indice]);
    }
}
```



## Osservazioni

Assegnazione tra array:

Dati i seguenti array:

```
typedef int anArray[10];  
anArray    Array1, Array2;
```

l'istruzione:

```
Array2 = Array1;
```

**E' scorretta!**

Sarà necessaria un'istruzione ciclica che "scorra" i singoli elementi dell'array

## I tipi strutturati: il costruttore **struct**

Tipo *impiegato*: *nome, cognome, codice fiscale, indirizzo, numero di telefono, eventuali stipendio, data di assunzione* e via di seguito.

Tipo *famiglia*: un certo insieme di *persone*, un *patrimonio*, costituito a sua volta da un insieme di *beni*, ognuno con un suo *valore*, un *reddito* annuo, *spese* varie, ...

Queste strutture informative sono *eterogenee*: l'array non si presta a questo tipo di aggregazione.

Il costruttore **struct**, invece, permette di definire una **struttura di dati** aggregando elementi anche eterogenei (di tipi diversi). I vari elementi si dicono **campi** della struttura.

Come il costruttore di *array*, anche il costruttore **struct** può essere usato per definire un tipo o direttamente per definire una variabile strutturata.

Il costruttore di record (parola chiave **struct** in C) è la risposta a questo tipo di esigenze.

# VARIABILI STRUTTURATE: il costruttore **STRUCT**

## *Esempi*

```
typedef char String[50];
```

```
typedef struct {    int    Giorno;  
                  int    Mese;  
                  int    Anno;  
                }    Data;
```

```
typedef struct {    String  Destinatario;  
                  int      Importo;  
                  Data      DataEmissione;  
                }    DescrizioneFatture;
```

```
typedef enum {Dirigente, Impiegato, Operaio} CatType;
```

```
typedef struct {    String  Nome;  
                  String  Cognome;  
                  int      Stipendio;  
                  char      CodiceFiscale[16];  
                  Data      DataAssunzione;  
                  CatType  Categoria;  
                }    Personale;
```

## Dichiarazione di variabili

La dichiarazione di variabili procede poi come al solito:

```
Personale  Dip1, Dip2;
```

Oppure è possibile definire il nuovo tipo in forma *anonima* e nello stesso tempo dichiarare le variabili:

```
struct {    String Nome;  
            String Cognome;  
            int Stipendio;  
            char    CodiceFiscale[16];  
            Data    DataAssunzione;  
            CatType Categoria;  
        }    Dip1, Dip2;
```

## Le accortezze nel gestire le strutture

Per accedere ai singoli campi di una struttura si usa la cosiddetta “**dot notation**”.

*esempi*

```
Dip1.data_assunzione.anno = 1998;  
Dip2.stipendio = Dip1.stipendio * 1.1;
```

Si possono costruire array i cui elementi sono variabili strutturate.

*esempi*

```
Personale      Dip [300];  
Dip [20].stipendio = Dip1.stipendio;
```

Oltre alle operazioni sui singoli campi, una variabile strutturata prevede **operazioni globali**.

*Esempi*

```
Dip1 = Dip2;
```

**E' lecito** e fa esattamente ciò che ci si aspetta: copia l'intera struttura Dip1 in Dip2, **comprese le sue componenti che sono costituite da array!**

Il perché di questa stranezza risiede nel modo in cui in C sono realizzati gli array.

## Il costruttore **puntatore**

Esso permette di costruire il ***tipo puntatore ad un tipo di oggetto*** (variabile semplice o strutturata).

### Dichiarazione di tipo puntatore:

```
typedef      tipo      *punt;
```

Ciò consente di dichiarare variabili di tipo puntatore:

```
punt          P, Q;
```

### Dereferenziazione:

\*P indica il valore della cella di memoria il cui indirizzo è contenuto in P

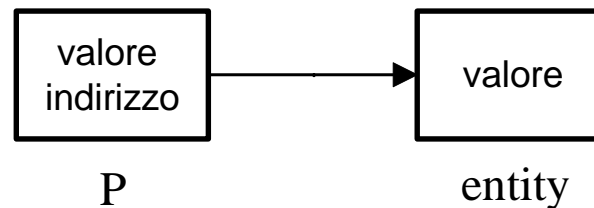
## Puntatori e tipi

Una variabile di tipo puntatore può far riferimento ad un elemento del tipo indicato nella costruzione.

```
tipoelemento      entity, s, t;
```

A variabili di tipo puntatore può essere assegnato un valore tramite l'**operatore unario &** (indirizzo di).

```
P = &entity;
```



L'operatore unario & significa "indirizzo di" ed è il duale dell'operatore '\*'.

```
typedef      TipoDato      *TipoPuntatore;  
TipoPuntatore      P, Q;  
TipoDato      y, z;  
    P = &y;  
    Q = &z;  
    P = Q;
```

y e z sono di tipo TipoDato mentre P e Q sono *puntatori* a variabili di tipo TipoDato.

# VARIABILI STRUTTURATE: il costruttore **puntatore**

## Puntatori e tipi

```
typedef TipoDato      *TipoPuntatore;  
typedef AltroTipoDato *AltroTipoPuntatore;  
TipoDato              *Puntatore;  
TipoDato              **DoppioPuntatore;  
TipoPuntatore         P, Q;  
AltroTipoPuntatore    P1, Q1;  
TipoDato              x, y;  
AltroTipoDato         z, w;
```

### istruzioni corrette:

```
Puntatore = &y;  
DoppioPuntatore = &P;  
Q1 = &z;  
P = &x;  
P = Q;  
*P = *Q;  
*Puntatore = x;  
P = *DoppioPuntatore;  
z = *P1;  
Puntatore = P;
```

### istruzioni scorrette:

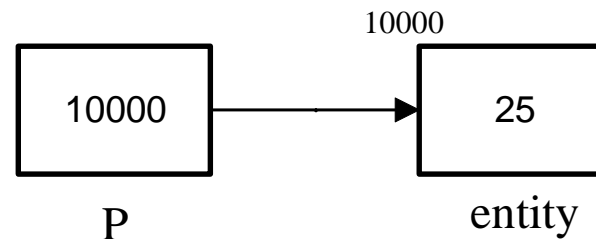
```
P1 = P; (warning)  
w = *P; (error)  
*DoppioPuntatore = y; (warning)  
Puntatore = DoppioPuntatore; (warning)  
*P1 = *Q; (error)
```



## VARIABILI STRUTTURATE: il costruttore **puntatore**

*esempio*

Se P è una variabile puntatore che si riferisce ad una entità di nome entity (supposta di tipo intero), valore 25 e indirizzo di memoria 10000, allora il valore di P sarà 10000.



### **Riassumendo e completando**

Operazioni applicabili a variabili puntatori:

- ⇒ assegnamento dell'indirizzo di una variabile tramite l'operatore unario &;
- ⇒ assegnamento del valore di un altro puntatore;
- ⇒ assegnamento del valore speciale NULL. Se una variabile puntatore ha valore NULL, \*P è indefinito: P non punta ad alcuna informazione significativa. l'operazione di dereferenziazione, indicata dall'operatore \*;il confronto basato sulle relazioni ==, !=, >, <, <=, >=;operazioni aritmetiche;l'assegnamento di indirizzi di memoria a seguito di operazioni di allocazione esplicita di memoria

## Array e puntatori

Definiamo un record e dichiariamo una variabile puntatore:

```
typedef struct {   int    PrimoCampo;  
                  char    SecondoCampo;           }   TipoDato;
```

```
TipoDato    x, *P;
```

```
P = &x;
```

Accesso al campo PrimoCampo di x, attraverso il puntatore P, usando la *dot notation*:

```
(*P).PrimoCampo = 12;           /* Inserisce 12 nel campo PrimoCampo di x */
```

Esiste una sintassi abbreviata:

```
P->PrimoCampo = 12;           /* Inserisce 12 nel campo PrimoCampo di x */
```

## VARIABILI STRUTTURATE: il costruttore **puntatore**

**Gli elementi di un array occupano posizioni di memoria contigue**

### **L'operatore `sizeof`**

L'operatore **`sizeof`** produce il numero di byte occupati da ciascun elemento di un array o da un array nel suo complesso.

Se si usano quattro byte per la memorizzazione di un valore **`int`**:

```
int a[5]; allora:
```

```
sizeof(a[2])
```

restituisce il valore 4 e:

```
sizeof(a)
```

restituisce il valore 20.

**Il nome di una variabile di tipo array viene considerato in C come l'indirizzo della prima parola di memoria che contiene il primo elemento della variabile di tipo array (lo 0-esimo ...).**

Se ne deduce:

che `a` "punta" a una parola di memoria esattamente come un puntatore;

che `a` punta sempre al primo elemento della variabile di tipo array (è un puntatore "fisso" al quale non è possibile assegnare l'indirizzo di un'altra parola di memoria).

## Operazioni di somma e sottrazione su puntatori

Se  $p$  e  $a$  forniscono l'indirizzo di memoria di elementi di tipo opportuno,  $p+i$  e  $a+i$  forniscono l'indirizzo di memoria dell' $i$ -esimo elemento successivo di quel tipo.

Se  $i$  è una variabile intera:

la notazione  $a[i]$  è equivalente a  $*(a+i)$

Analogamente, se  $p$  è dichiarato come puntatore a una variabile di tipo **int**:

la notazione  $p[i]$  è equivalente a  $*(p+i)$ .

Ne segue che:

$p = a$  è equivalente a  $p = \&a[0];$

$p = a+1$  è equivalente a  $p = \&a[1];$

Mentre non sono ammessi assegnamenti ad  $a$  del tipo:

$a = p;$

$a = a + 1;$

Se  $p$  e  $q$  puntano a due diversi elementi di un array,  $p-q$  restituisce un valore intero pari al *numero di elementi* esistenti tra l'elemento cui punta  $p$  e l'elemento cui punta  $q$ .

*Non la differenza tra il valore dei puntatori.*

Se il risultato di  $p-q$  è pari a 3 e supponendo che ogni elemento dell'array sia memorizzato in 4 byte, la differenza tra l'indirizzo contenuto in  $p$  e l'indirizzo contenuto in  $q$  darebbe 12.

## Attenzione ai “rischi” dei puntatori

Effetti collaterali (*side effects*):

```
*P = 3;  
*Q = 5;  
P = Q;  
/* a questo punto *P = 5 */  
*Q = 7;
```

A questo punto  $*Q = 7$ , ma anche  $*P = 7$

Un assegnamento esplicito alla variabile puntata da Q determina un assegnamento nascosto alla variabile puntata da P.

Caso particolare di *aliasing*, ovvero del fatto che uno stesso oggetto viene identificato in due modi diversi.

## Errori a Compile-time ed errori a Run-time

### Puntatori e tipizzazione delle variabili puntate:

Viene *segnalato* dal compilatore il tentativo di utilizzo congiunto di puntatori dichiarati come puntanti a dati di tipo differente

### Tipizzazione “forte” del C

Il C consente di eseguire tutte le espressioni e le assegnazioni che coinvolgono variabili di tipo diverso, a condizione che:

- i tipi siano compatibili
- vengano applicate le *regole di conversione implicita*.

Poiché il compilatore verifica le due condizioni esposte, si dice che il linguaggio persegue l'obiettivo della tipizzazione forte.

## Altri errori “a compile-time”

- Errato annidamento di parentesi
- Mancata o errata dichiarazione di variabile
- ...

## Errori a run-time

- Divisione per 0
- Indice di un array fuori dai limiti
- Accesso ad una variabile non inizializzata
- ...