

Studio sistematico del linguaggio C: tipi di dati

Studio sistematico

- Intraprendiamo ora lo studio sistematico degli elementi essenziali del linguaggio C
- Lo scopo non è tanto diventare esperti del linguaggio, quanto impossessarsi di alcuni concetti fondamentali della programmazione che dovranno essere applicati in vari contesti e usando diversi strumenti/linguaggi
- C sarà perciò soltanto un punto di riferimento ma dovremo presto imparare a “giostrare tra vari linguaggi”
- Alcuni di essi, i più tradizionali (Ada, Pascal, Modula-2, C++, Java,), hanno molto in comune con C e ad essi soprattutto faremo riferimento
- Altri sono basati su principi meno convenzionali, che verranno illustrati in corsi successivi

Il “vero” linguaggio C

- I programmi visti finora non sono ancora “puro codice C”
- Che cosa manca per arrivare al “puro codice C”?
- Ricominciamo da capo (quasi)...

```
main() {  
    int numero, somma;  
    somma = 0;  
    scanf("%d", &numero);  
    while (numero != 0) {  
        somma = somma + numero;  
        scanf("%d", &numero);  
    }  
    printf("La somma dei numeri digitati è: %d\n", somma);  
}
```

La struttura dei programmi C

- Un programma C deve contenere, nell'ordine:
 - l'identificatore predefinito main seguito dalle parentesi ()
 - già visto in precedenza
 - due parti, sintatticamente racchiuse dalle parentesi {}
 - la parte **dichiarativa**
 - la parte **esecutiva**
- La parte **dichiarativa** di un programma costituisce la principale novità
- Essa elenca tutti gli elementi che fanno parte del programma, con le loro principali caratteristiche
- La parte **esecutiva** consiste in una successione di istruzioni come già descritto in pseudo-C

La parte dichiarativa

- Tutto ciò che viene usato va dichiarato
- In prima istanza: altri elementi seguiranno:
 - dichiarazione delle **costanti**
 - dichiarazione delle **variabili**
- Perché obbligare il programmatore a questa fatica ... inutile?
 - aiuta la diagnostica (ovvero segnalazione di errori):
 - $x = \text{alfa};$
 - $\text{alba} = \text{alfa} + 1;$
 - senza dichiarazione alba è una nuova variabile
- Principio importante: meglio un po' più di fatica nello scrivere un programma che nel leggerlo e capirlo!

Dichiarazione delle variabili

- Una dichiarazione di variabile consiste in uno **specificatore di tipo**, seguito da una lista di uno o più identificatori di variabili separati da una virgola
 - ogni dichiarazione termina con ‘;’
- Il concetto di **tipo** (di dato, o informazione) è un concetto fondamentale della programmazione
- La stessa stringa di bit può rappresentare un intero, un carattere, ...
- Il tipo di dato è il “ponte” tra la visione astratta dell’informazione e la sua rappresentazione concreta: strumento fondamentale di **astrazione**
- Abbiamo già fatto uso del concetto di tipo (di una variabile): abbiamo infatti parlato di interi, caratteri, ... tipi strutturati
- A poco a poco saremo sempre più sistematici nell’esposizione del sistema di tipi (del C e, in generale, nella programmazione)

Cos'è l'astrazione?

“The purpose of abstraction is **not** to be **vague**, but to create a new semantic level in which one can be **absolutely precise**.”

Edsger Dijkstra



J. Kramer, “Is Abstraction the Key to Computing?”, Comm. of the ACM, 2007.

Tre tipi già noti

- Le parole chiave **int**, **float**, **char** specificano, rispettivamente i tipi intero, reale, carattere:

```
float  x,y;  
int    i,j;  
char  simb;
```

- Equivalenti a:

```
float  x;  
int    i,j;  
char  simb;  
float  y;
```

- Se un identificatore di variabile **x** è dichiarato di tipo **int**, esso potrà essere usato nella parte esecutiva solo come tale
 - di conseguenza, **x** non assumerà mai un valore reale, come per esempio 3.14

Costanti

- Una dichiarazione di costante associa **permanentemente** un valore a un identificatore
- Come nel caso della dichiarazione di variabili, la sezione della dichiarazione di costanti consiste in una lista di dichiarazioni
- Ogni dichiarazione, a sua volta, consiste in:
 - la parola chiave **const**;
 - lo **specificatore di tipo**;
 - l'identificatore della costante;
 - il simbolo =;
 - il valore della costante
 - il solito "terminatore" ';' ;
- Esempi di dichiarazioni di costanti:

const	float	PiGreco = 3.14;
const	float	PiGreco = 3.1415, e = 2.718;
const	int	N = 100, M = 1000;
const	char	CAR1 = 'A', CAR2 = 'B';
- Un eventuale assegnamento a una costante sarebbe segnalato come errore

Perché dichiarare le costanti?

- L'istruzione:
 $\text{AreaCerchio} = \text{PiGreco} * \text{RaggioCerchio} * \text{RaggioCerchio};$

è equivalente a:
 $\text{AreaCerchio} = 3.14 * \text{RaggioCerchio} * \text{RaggioCerchio};$
- Maggior astrazione
 - l'area del cerchio è il prodotto del quadrato del raggio per π , e non “per 3.14”, né “3.1415”, né “3.14159” ecc.
 - se non mi accontento più di 2 cifre decimali per approssimare π , mi basta modificare la sua dichiarazione, non tutte le istruzioni in cui ne faccio uso
 - altri esempi di parametricità ottenuta attraverso la dichiarazione di costanti saranno illustrati in seguito

Parte esecutiva

- L'unica differenza tra il codice “pseudo-C” e il codice “C reale” sta nelle istruzioni di I/O
- Le “istruzioni” di I/O non sono vere e proprie istruzioni del linguaggio, ma sottoprogrammi di libreria, concetto che verrà illustrato più avanti
- printf richiede una **stringa di controllo** e un insieme di **elementi da stampare**:

printf(stringa di controllo, elementi da stampare);
- La **stringa di controllo** è una stringa che viene stampata in uscita e può contenere caratteri detti **di conversione** o **di formato** preceduti dal simbolo %
- I caratteri di formato %d, %f, %c, %s provocano la stampa sullo Standard Output rispettivamente:
 - di un numero intero decimale, di un numero reale, di un carattere, di una stringa di caratteri
 - il simbolo \n nella stringa di controllo provoca un salto a nuova riga per la successiva scrittura
- L'insieme degli **elementi da stampare** è una lista di variabili, di costanti o di espressioni composte con variabili e costanti

Un primo esempio

- `printf ("Lo stipendio annuo dei dipendenti di categoria %d è pari a E. %f", cat_depend, stip_medio);`
 - se `cat_depend` è una variabile di tipo `int` che rappresenta la categoria dei dipendenti considerati, l'istruzione `printf` viene eseguita usando il suo valore corrente, per esempio 6
 - se `stip_medio` è una variabile di tipo `float` che rappresenta lo stipendio medio calcolato per il gruppo di dipendenti considerato, l'istruzione `printf` viene eseguita usando il suo valore corrente, per esempio 1570580.0
- L'esecuzione dell'istruzione `printf` provocherà la stampa sullo schermo della frase

Lo stipendio annuo dei dipendenti di
categoria 6 è pari a E. 1570580.0

Un secondo esempio

- `printf("%s\n%c%c\n\n%s\n", "Questo programma è stato scritto da", iniz_nome, iniz_cognome, "Buon lavoro!");`
 - se `iniz_nome` è una variabile di tipo `char` che rappresenta l'iniziale del nome del programmatore, l'istruzione `printf` viene eseguita usando il suo valore corrente, per esempio `G`
 - se `iniz_cognome` è una variabile di tipo `char` che rappresenta l'iniziale del cognome del programmatore, l'istruzione `printf` viene eseguita usando il suo valore corrente, per esempio `M`

- L'esecuzione dell'istruzione `printf` produce la visualizzazione di:

```
Questo programma è stato scritto da  
GM
```

```
Buon lavoro!
```

- Notare l'effetto dei simboli `\n`

Input di informazioni

- Anche scanf è del tipo

scanf(stringa di controllo, elementi da leggere);

- ...ma i nomi delle variabili sono preceduti dall'operatore unario &

Un primo esempio

- `scanf("%c%c%c%d%f", &c1, &c2, &c3, &i, &x);`
- Se al momento dell'esecuzione dell'istruzione `scanf` l'utente inserisce i seguenti dati:
- `ABC 3 7.345`
 - la variabile `c1` (di tipo `char`) assume il valore `A`
 - la variabile `c2` (di tipo `char`) assume valore `B`
 - la variabile `c3` (di tipo `char`) assume valore `C`
 - la variabile `i` (di tipo `int`) assume valore `3`
 - la variabile `x` (di tipo `float`) assume valore `7.345`

#include

- ogni programma che utilizza al suo interno le funzioni printf e scanf deve dichiarare l'uso di tali funzioni nella parte direttiva che precede il programma principale.

#include <stdio.h>

- E' una direttiva data a una parte del compilatore, chiamata preprocessore, che include una copia del contenuto del file stdio.h. Essa provoca l'inclusione (una copia) del contenuto del file stdio.h. Tra le dichiarazioni di funzione contenute in stdio.h sono presenti quelle di printf e scanf. Questo consente una esatta compilazione del programma che utilizza nella sua parte eseguibile printf e scanf e una produzione corretta del codice eseguibile grazie all'utilizzo del codice che il sistema C mette a disposizione per printf e scanf.
- E finalmente ...

Un programma completo

```
main() {  
    int a, b, somma;  
    printf ("inserisci come valore dei due addendi due numeri interi\n");  
    scanf ("%d%d", &a, &b);  
    somma = a + b;  
    printf ("La somma di a+b è:\n%d \nArrivederci!\n", somma);  
}
```

Se vengono inseriti i dati 3 e 5 l'effetto dell'esecuzione del programma sullo Standard Output è il seguente:

```
La somma di a+b è:  
8  
Arrivederci!
```

Se invece fossero stati omessi i primi due simboli \n nella stringa di controllo?

Il programma è anche un primo esempio di **programma interattivo!**

Tipo di dato

- Ci siamo già imbattuti in questo concetto:
 - tipo di dato = tipo di informazione
 - alcuni tipi: interi, reali, caratteri, array, ...
 - ma anche: fatture, conti correnti, pagine web, ...
 - quanti tipi? ...infiniti! ...e allora?
- Definizione generale di tipo di dato
 - insieme di valori e di operazioni ad esso applicabili
- **Tipo di dato astratto:**
 - conta la **visione esterna**, non la rappresentazione interna, ovvero il punto di vista di chi usa il tipo, non di chi lo realizza
 - gli interi, e tutti gli altri tipi, all'interno di un calcolatore non sono che sequenze di bit
 - l'hardware ci permette di astrarre da questa conoscenza: ci permette di scrivere $23, 555, ..$ e $74 + 223$ senza conoscere quale algoritmo viene applicato per calcolare la somma

Classificazione dei tipi di dato

- Tipi **semplici** (interi, caratteri, reali, ...)
 - valori del tipo contenuti in una cella di memoria
 - valori vengono gestiti dalle varie operazioni in modo unitario
- Tipi **strutturati** (per ora: array)
 - valori del tipo contenuti in diverse celle di memoria
 - valori possono essere scomposti in elementi più semplici trattabili separatamente
- Tipi **predefiniti** (nel linguaggio) (interi, caratteri, ...)
- Tipi definiti dall'utente (per soddisfare le infinite e imprevedibili esigenze)
 - in C: età, data, conto_corrente non sono predefiniti: devono essere costruiti dal programmatore
 - in altri linguaggi “special purpose” potrebbero esserlo (e.g. data nei fogli elettronici)
- Il concetto è generale: l'attuazione varia a seconda del linguaggio

Tipi semplici predefiniti

- Quattro tipi di base: char (caratteri), int (interi), float (reali), double (reali in precisione doppia)
 - i “qualificatori di tipo” signed o unsigned possono essere applicati a char e a int
 - i qualificatori short o long a int
 - il qualificatore long a double
- Alcuni esempi:
 - char
 - signed char
 - unsigned char
 - signed short int signed short, short
 - signed int signed, int
 - signed long int long int, signed long, long
 - unsigned short int unsigned short
 - unsigned int unsigned
 - unsigned long int unsigned long
 - float
 - double
 - long double

Il tipo int

- Il tipo “interi” non è l’insieme {..., -1, 0, 1, 2, ...}, bensì quello stesso insieme “corredato” di somma, sottrazione, moltiplicazione ecc...
 - il tipo “interi” della matematica ha infiniti valori, e anche infinite operazioni
 - non così in C: int è **un’approssimazione finita** del corrispondente tipo matematico, legata alla finitezza della memoria reale del calcolatore
- Vale sempre che
spazio allocato (short int) \leq spazio allocato (int) \leq spazio allocato(long int)
- Un signed int usa un bit per la rappresentazione del segno
- Un unsigned int utilizza tutti gli n bit per rappresentare il valore intero supposto positivo
- La particolare codifica di int è predefinita dalla definizione del linguaggio
 - INT_MIN e INT_MAX lo sono dall’implementazione del linguaggio, non dalla definizione dello stesso
- Su qualunque macchina venga eseguito un programma vale comunque la proprietà seguente
 - spazio allocato (signed int) = spazio allocato (unsigned int)

Operazioni per dati di tipo int

- = Assegnamento di un valore int a una variabile int
 - + Somma (tra int ha come risultato un int)
 - Sottrazione (tra int ha come risultato un int)
 - * Moltiplicazione (tra int ha come risultato un int)
 - / Divisione con troncamento della parte non intera (risultato int)
 - % Resto della divisione intera
 - == Relazione di uguaglianza
 - != Relazione di diversità
 - < Relazione “minore di”
 - > Relazione “maggiore di”
 - <= Relazione “minore o uguale a”
 - >= Relazione “maggiore o uguale a”
-
- Se alcune operazioni fornissero un risultato non appartenente all’insieme dei valori consentito (es. se il risultato di una moltiplicazione maggiore di INT_MAX) il risultato sarebbe una segnalazione di errore (Integer Overflow)
 - in questo caso il risultato concreto non corrisponde al valore astratto!

I tipi float e double

- Approssimazione dei **numeri reali**, non solo dal punto di vista del loro limite, ma anche della precisione di rappresentazione
- Due diverse rappresentazioni:
 - Rappresentazione decimale, o in **virgola fissa**
 - 3.14 1234.543 328543.4 0.000076
 - rappresentare 10^{-900} in questa maniera richiederebbe un enorme spreco di cifre, e quindi di celle di memoria
 - Rappresentazione in **virgola mobile (float-ing point)**
 - mantissa ed esponente (della base 10)
 - se un numero n ha mantissa m ed esponente e , il suo valore è $n = m \times 10^e$
 - 1780000.0000023 può essere rappresentato in virgola mobile nei modi seguenti:
178000.00000023E1 178000000000023E-7
- Le notazioni sono interscambiabili e la macchina provvede automaticamente alle necessarie conversioni
- Vale sempre che
spazio allocato (float) \leq spazio allocato (double)
 \leq spazio allocato (long double)
- Nell standard IEEE 754 che abbiamo già visto, float (precisione singola) ha 32 bit, double (precisione doppia) ha 64 bit

Operazioni per dati di tipo float e double

=	Assegnamento
+	Somma
-	Sottrazione
*	Moltiplicazione
/	Divisione (a risultato reale)
==	Relazione di uguaglianza
!=	Relazione di diversità
<	Relazione “minore di”
>	Relazione “maggiore di”
<=	Relazione “minore o uguale a”
>=	Relazione “maggiore o uguale a”

Alcune note

- Sia internamente (in base 2) che esternamente (in base 10) le due rappresentazioni in virgola fissa e virgola mobile sono interscambiabili
 - esistono **algoritmi interni di conversione** che operano in maniera trasparente
- La libreria standard fornisce anche diverse funzioni matematiche predefinite (sqrt, pow, exp, sin, cos, tan...)
 - per ora trattiamole come operazioni normali built-in (però occorre includere l'opportuno file)
- Attenzione agli **arrotondamenti**:

$(x/y) * y == x$
potrebbe risultare falsa !

Invece di scrivere
if (x == y) ...
è meglio scrivere
if (x <= y + .000001 && y <= x + .000001) ...

Il tipo char

- L'insieme dei dati di tipo char è l'insieme dei caratteri ASCII, e contiene tutte le lettere, le cifre e i simboli disponibili sulle normali tastiere
 - la codifica ASCII consente la rappresentazione di ciascun carattere attraverso un **opportuno valore intero**
 - definisce inoltre l'**ordinamento** dei valori: per qualsiasi coppia di caratteri x e y, $x < y$ se e solo se x precede y nell'elenco dei caratteri
- Alcuni caratteri sono **caratteri di controllo**: la loro scrittura non consiste in una vera e propria stampa di un simbolo sulla carta o sul video, ma nell'esecuzione di un'operazione correlata alla visualizzazione dei dati:
 - `\n` che provoca un a capo, `\b` = "backspace", `\t` = "horizontal tab"....

La tabella ASCII

0	<NUL>	32	<SPC>	64	@	96	`	128	Ä	160	†	192	¿	224	‡
1	<SOH>	33	!	65	A	97	a	129	Å	161	°	193	¡	225	·
2	<STX>	34	"	66	B	98	b	130	Ç	162	¢	194	¬	226	,
3	<ETX>	35	#	67	C	99	c	131	É	163	£	195	√	227	„
4	<EOT>	36	\$	68	D	100	d	132	Ñ	164	§	196	f	228	‰
5	<ENQ>	37	%	69	E	101	e	133	Ö	165	•	197	≈	229	Â
6	<ACK>	38	&	70	F	102	f	134	Ü	166	¶	198	Δ	230	Ê
7	<BEL>	39	'	71	G	103	g	135	á	167	ß	199	«	231	Á
8	<BS>	40	(72	H	104	h	136	à	168	®	200	»	232	Ë
9	<TAB>	41)	73	I	105	i	137	â	169	©	201	...	233	È
10	<LF>	42	*	74	J	106	j	138	ä	170	™	202		234	Í
11	<VT>	43	+	75	K	107	k	139	ã	171	'	203	À	235	Î
12	<FF>	44	,	76	L	108	l	140	å	172	..	204	Ã	236	Ï
13	<CR>	45	-	77	M	109	m	141	ç	173	≠	205	Ö	237	Ì
14	<SO>	46	.	78	N	110	n	142	é	174	Æ	206	Œ	238	Ó
15	<SI>	47	/	79	O	111	o	143	è	175	Ø	207	œ	239	Ô
16	<DLE>	48	0	80	P	112	p	144	ê	176	∞	208	–	240	🍏
17	<DC1>	49	1	81	Q	113	q	145	ë	177	±	209	—	241	Ò
18	<DC2>	50	2	82	R	114	r	146	í	178	≤	210	“	242	Ú
19	<DC3>	51	3	83	S	115	s	147	ì	179	≥	211	”	243	Û
20	<DC4>	52	4	84	T	116	t	148	î	180	¥	212	`	244	Ü
21	<NAK>	53	5	85	U	117	u	149	ï	181	μ	213	'	245	ı
22	<SYN>	54	6	86	V	118	v	150	ñ	182	ð	214	÷	246	^
23	<ETB>	55	7	87	W	119	w	151	ó	183	Σ	215	◊	247	~
24	<CAN>	56	8	88	X	120	x	152	ò	184	Π	216	ÿ	248	—
25		57	9	89	Y	121	y	153	ô	185	π	217	Ŷ	249	˘
26	<SUB>	58	:	90	Z	122	z	154	ö	186	∫	218	/	250	˙
27	<ESC>	59	;	91	[123	{	155	õ	187	ª	219	€	251	◦
28	<FS>	60	<	92	\	124		156	ú	188	º	220	<	252	¸
29	<GS>	61	=	93]	125	}	157	ù	189	Ω	221	>	253	”
30	<RS>	62	>	94	^	126	~	158	û	190	æ	222	fi	254	˚
31	<US>	63	?	95	_	127		159	ü	191	ø	223	fl	255	˛

Il tipo char

- Il compilatore ANSI C alloca per una variabile di tipo char 1 byte e lo stesso avviene per una variabile di tipo signed char o unsigned char
- Un byte può contenere la codifica binaria di 256 valori differenti
- Per un signed char l'insieme dei valori va da -128 a +127, per un unsigned char l'insieme dei valori va da 0 a 255 (ASCII esteso e problemi di standardizzazione ...)
- Sono definite le operazioni di assegnamento (=), le operazioni aritmetiche (+, -, *, /, %) e quelle relazionali (==, !=, < ecc.)
- La comunanza di operazioni tra char e int è una naturale conseguenza della rappresentazione dei caratteri tramite numero intero

Un tipico esempio delle caratteristiche del C

- Problema:
 - leggere una sequenza di caratteri (terminata da #)
 - per ciascun carattere letto viene stampato il relativo codice ASCII
 - nel caso sia una lettera dell'alfabeto minuscola, viene operata la trasformazione in lettera maiuscola

Codice

```
main() {  
    char C, CM;  
    printf("Inserire un carattere – # per terminare il programma\n");  
    scanf(" %c", &C);  
    while (C != '#') {  
        printf("Il codice ASCII del carattere %c è %d\n", C, C);  
  
        if (C >= 'a' && C <= 'z') {  
  
            CM = C - ('a' - 'A');  
            printf("La lettera maiuscola per %c è %c e il suo codice ASCII è %d\n", C, CM, CM);  
        }  
  
        printf("Inserire un carattere – # per terminare il programma\n");  
        scanf(" %c", &C);  
    }  
}
```

Se il
carattere è
una lettera
minuscola...

La “differenza” tra ‘a’ e
‘A’ è lo scarto fra la
rappresentazione ASCII
delle lettere maiuscole e
minuscole

Stampiamo lo
stesso dato
con formati
diversi!

Nuovi tipi

- Una dichiarazione di consiste nella parola chiave `typedef` seguita dalla rappresentazione o costruzione del nuovo tipo, dall'identificatore del nuovo tipo, e dal simbolo “;” che chiude la dichiarazione

```
typedef int  anno;
```

- Una volta definito e identificato un nuovo tipo ogni variabile può essere dichiarata di quel tipo come di ogni altro tipo già esistente

```
char    x;  
anno     y;
```

- `typedef` non consente di costruire veri e propri nuovi tipi astratti (mancano le operazioni ... colmeremo più tardi questa lacuna)

Ridefinizione

- Sintassi generale

typedef TipoEsistente NuovoTipo;

- TipoEsistente può essere sia un tipo built-in (predefinito), per esempio int, sia un tipo user-defined precedentemente definito:

```
typedef   int    tipo1;  
typedef   char   tipo2;  
typedef   tipo1  tipo3;  
typedef   tipo2  tipo4;
```


Enumerazione esplicita dei valori

- Un nuovo tipo può essere costruito anche elencando (ossia, **enumerando**), all'interno di una coppia di parentesi graffe e separati tra loro da una virgola, tutti i suoi valori:

```
typedef enum    {lun, mar, mer, gio, ven, sab, dom} GiornoDellaSettimana;  
typedef enum    {rosso, verde, giallo, violetto, marrone, nero, ocra} colore;  
typedef enum    {Giovanni, Claudia, Carla, Simone, Serafino} persone;  
typedef enum    {gen, feb, mar, apr, mag, giu, lug, ago, set, ott, nov, dic} mese;
```

- Data la seguente dichiarazione di variabili:

```
persone individuo, individuo1, individuo2;
```

- E' possibile scrivere istruzioni del tipo

```
individuo = Giovanni;  
if (individuo1 == individuo2) individuo = Claudia;
```

- Senza includere i valori Giovanni e Claudia tra virgolette
 - infatti, non sono dei valori di tipo stringa!

Alcune osservazioni

- Spesso i valori del nuovo tipo sono rappresentati da nomi; però il sistema associa comunque a tali nomi un **progressivo valore intero**
 - per esempio, una variabile *x* dichiarata di tipo mese come prima che assume durante l'esecuzione del programma valore gen assume in realtà valore 0, mentre assume valore 3 per apr, ...
- Mancanza di astrazione del C: consente l'uso della **rappresentazione interna** dei valori all'interno di enumerazioni
- Di conseguenza, le operazioni applicabili sono le stesse degli interi
 - operazioni aritmetiche (+, −, *, /, %), assegnamento (=), confronto per uguaglianza (==), diversità (!=), precedenza stretta e debole (<, <=, >, >=)
 - in particolare, la relazione di precedenza è definita dall'ordine in cui vengono elencati i valori del tipo


(apr < giu)
(rosso < arancio)
producono un risultato un intero diverso da 0 (valore logico "true")

(dom < lun)
(Simone < Giovanni)
producono 0 (corrispondente al valore "false"):
- Un tipo costruito mediante enumerazione è un tipo totalmente ordinato, limitato, ed enumerabile

Caso particolare

- In C la definizione di variabili che possano assumere valore false o true (**variabili Booleane**) richiede la dichiarazione di un tipo tramite il costruttore di tipo enum:

```
typedef enum {false, true} boolean;  
boolean flag, ok;
```



Attenzione
all'ordine di
apparizione
nella enum!

- flag e ok possono così essere definite come variabili in grado di assumere valore vero (true) o falso (false) durante l'esecuzione di un programma che le usi
- Altri linguaggi hanno il tipo predefinito Boolean

Tipi strutturati

- Anche se in passato abbiamo introdotto gli array come variabili di “tipo strutturato” array, ad essere rigorosi, in C i tipi predefiniti strutturati non esistono
- Questa affermazione può sembrare in contrasto con la seguente tipica dichiarazione di array in C:

```
int lista[20];
```
- ...che dichiara un array di nome lista costituito da 20 interi (quindi il suo indice assume i valori da 0 a 19)
- Esaminiamo allora dalla radice il concetto di tipo strutturato
- In primis, i tipi strutturati sono solo ottenuti mediante costruttori di tipo, ossia sono tutti costruiti dal programmatore attraverso alcuni meccanismi fondamentali

Torniamo agli array...

- Costruzione del tipo:
typedef int anArray[20];
- Dichiarazione di variabili di tipo anArray:
anArray lista1, lista2;
- La dichiarazione
int lista[20];
- ...va perciò interpretata come un'abbreviazione per
typedef int arrayAnonimo[20];
arrayAnonimo lista
- L'array è un costruttore di tipo, non un tipo:
typedef int anArray [20]; e
typedef double NuovaLista[30];
- ...danno luogo a due tipi diversi!

Quando esplicitare il nome del tipo?

- Cosa è più chiaro e leggibile tra:

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

oppure

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

- Cosa è più chiaro e leggibile tra:

```
typedef double PioggeMensili[12];  
PioggeMensili Piogge01, Piogge02, Piogge03;
```

```
typedef double IndiciBorsa[12];  
IndiciBorsa Indici01, Indici02, Indici03;
```

oppure

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

Array di array

- È quindi utile avere anche “array di array”:
typedef int Vettore[20];
typedef Vettore MatriceIntera20Per20[20];
MatriceIntera20Per20 matrice1;
- Altri esempi:
typedef int Matrice20Per20[20][20];
int matrice1[20][20];
int matricetrid1[10][20][30];
- Per accedere agli elementi di matricetrid1
potremmo scrivere: matricetrid1[2][8][15]

Non dimentichiamoci di von Neumann!

- Dall'array **multidimensionale** del C
- All'array **monodimensionale** della macchina di von Neumann
- Le n dimensioni vengono "linearizzate"

j \ i	0	1	2	3
	0	1	2	3
0	23	12	3	55
1	16	2	45	18
2	...			

Cella	Indirizzo
23	1000
12	1001
3	1002
55	1003
16	1004
2	...
45	$1000 + 4*j + i$
18
...	

Un brutto inconveniente

- Un array ha dimensioni fisse
 - gli estremi di variabilità degli indici di un array **non** possono cambiare durante l'esecuzione del programma!
- Ciò certamente riduce la flessibilità del linguaggio: ad esempio, consideriamo le stringhe di caratteri: siamo costretti a cose del tipo:

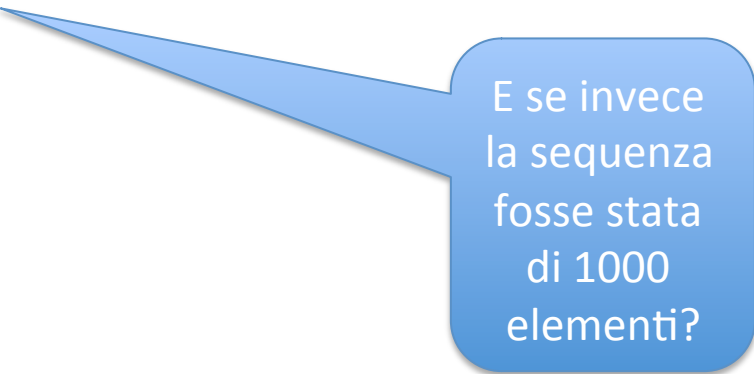
```
typedef char String[30];  
String Nome, Cognome;
```

- Parole corte memorizzate in tali variabili lascerebbero una notevole quantità di memoria (fisica) inutilizzata; d'altro canto, prima della lettura di un nuovo carattere, dovremmo anche prevedere istruzioni del tipo:

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

Controlli su lunghezza degli array

```
main() {  
    int  Contatore;  
    int  Memorizzazione[100];  
  
    Contatore = 0;  
    while (Contatore < 100) {  
        scanf("%d", &Memorizzazione[Contatore]);  
        Contatore = Contatore + 1;  
    }  
    Contatore = Contatore - 1;  
    while (Contatore >= 0) {  
        printf("%d\n", Memorizzazione[Contatore]);  
        Contatore = Contatore - 1;  
    }  
}
```



E se invece
la sequenza
fosse stata
di 1000
elementi?

La direttiva #define

```
#define LunghezzaSequenza 100
```

```
main() {  
    int Contatore;  
    int Memorizzazione[LunghezzaSequenza];  
  
    Contatore = 0;  
    while (Contatore < LunghezzaSequenza) {  
        scanf("%d", &Memorizzazione[Contatore]);  
        Contatore = Contatore + 1;  
    }  
    Contatore = Contatore - 1;  
    while (Contatore >= 0) {  
        printf("%d\n", Memorizzazione[Contatore]);  
        Contatore = Contatore - 1;  
    }  
}
```

Non si poteva ottenere la stessa cosa con const, in quanto LunghezzaSequenza viene utilizzata nella parte dichiarativa del programma!

Di nuovo: attenzione!

- Se Array1 e Array2 sono variabili definite nel modo seguente:

```
typedef int anArray[10];
```

```
anArray Array1, Array2;
```

- E' scorretta l'istruzione:

```
Array2 = Array1;
```

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

Problema

- Costruire un programma che legge due stringhe composte da esattamente 50 caratteri ciascuna e costruisce una terza stringa che concatena le prime due in ordine alfabetico
- Il programma stampa poi la stringa così creata

Una possibile soluzione

```
#define LunghezzaArray 50
```

```
main() {  
    int    i, j, k;  
    char   TempCar;  
    char   Array1[LunghezzaArray], Array2[LunghezzaArray];  
    char   ArrayConc[LunghezzaArray*2];  
  
    i = 0;  
    while (i < LunghezzaArray) {  
        scanf("%c", &TempCar);  
        Array1[i] = TempCar;  
        i = i + 1;  
    }  
    i = 0;  
    while (i < LunghezzaArray) {  
        scanf("%c", &TempCar);  
        Array2[i] = TempCar;  
        i = i + 1;  
    }  
}
```

Può essere
funzione di
una #define

Legge la prima e la seconda
stringa assicurandosi che non
superino i 50 caratteri...

...continua

```
i = 0;
while (i < LunghezzaArray && Array1[i] == Array2[i])
    i = i + 1;

if (i == LunghezzaArray || Array1[i] < Array2[i]) {
    k = 0; j = 0;
    while (j < LunghezzaArray) {
        ArrayConc[k] = Array1[j];
        k = k + 1;
        j = j + 1;
    }
    j = 0;
    while (j < LunghezzaArray){
        ArrayConc[k] = Array2[j];
        k = k + 1;
        j = j + 1;
    }
}
```

Confronta le due stringhe
carattere per carattere per
individuare quella che precede
in ordine alfabetico...

Le stringhe sono uguali oppure
la prima precede la seconda in
ordine alfabetico!

Concateniamo la
seconda alla
prima!

...e finisce!

```
else {  
    k = 0; j = 0;  
    while (j < LunghezzaArray) {  
        ArrayConc[k] = Array2[j];  
        k = k + 1;  
        j = j + 1;  
    }  
    j = 0;  
    while (j < LunghezzaArray) {  
        ArrayConc[k] = Array1[j];  
        k = k + 1;  
        j = j + 1;  
    }  
}  
  
k = 0;  
while (k < (LunghezzaArray*2)) {  
    printf("%c", ArrayConc[k]);  
    k = k + 1;  
}  
}
```

Altrimenti, la seconda stringa precede la prima in ordine alfabetico e le due vanno concatenate di conseguenza...

Stampiamo la concatenazione ottenuta in una delle due maniere.

Dati strutturati eterogenei

- Come definire un tipo di dato impiegato composto da nome, cognome, codice fiscale, indirizzo, numero di telefono, eventuali stipendio e data di assunzione e via di seguito?
- Una famiglia è costituita da un certo insieme di persone, da un patrimonio, costituito a sua volta da un insieme di beni, ognuno con un suo valore, da un reddito annuo, da spese varie, ...
- Queste strutture informative sono eterogenee: l'array non si presta a questo tipo di aggregazione
- Il costruttore **record** (parola chiave **struct** in C) è la risposta a questo tipo di esigenze

Alcuni esempi...

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

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



Ovviamente si assume che i tipi String e Data siano già stati precedentemente definiti...

Alcuni esempi...

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

dove CatType è ottenuto da **typedef enum** {Dirigente, Impiegato, Operaio} CatType;

La dichiarazione di variabili procede poi come al solito:

```
Dipendenti  Dip1, Dip2;
```

oppure senza ricorrere al typedef..

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

Alcune considerazioni

- L'accesso al campo di un record avviene mediante la dot notation

```
Dip1.Stipendio = Dip1.Stipendio + (Dip1.Stipendio*10) / 100;
```

- I meccanismi di accesso a elementi di variabili strutturate si possono combinare tra loro esattamente come si possono combinare i costruttori di tipi:

```
Dip1.DataAssunzione.Giorno = 3;  
Dip1.DataAssunzione.Mese = 1;  
Dip1.DataAssunzione.Anno = 1993;
```

- Se si vuole sapere se la prima lettera del cognome di Dip1 è A:

```
if (Dip1.Cognome[0] == 'A') ...
```

- Si vuole sapere se la fattura numero 500 è stata emessa entro il 1991 o no, e in caso affermativo, qual è il suo importo:

```
if (ArchivioFatture[500].DataEmissione.Anno <= 1991)  
    printf("%d", ArchivioFatture[500].Importo);  
else printf("La fattura in questione è stata emessa dopo il 1991\n");
```

Una stranezza del C

- Mentre non è permesso scrivere un assegnamento globale tra array ($a = b$)
- l'istruzione: $\text{Dip2} = \text{Dip1}$ è lecita 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 e potrà essere capito tra breve