Strutture e Tipi Built in

Informatica A AA 2023 / 2024

Giacomo Boracchi

6 Ottobre 2023

giacomo.boracchi@polimi.it

Warm up

Scrivere un programma che richiede due stringhe all'utente

Il programma controlla se le due stringhe contengono le stesse vocali nello stesso ordine

Hint: estrarre, da ogni stringa, una stringa contenente solo le vocali

Tipi di Dato in C

Tipi di dato

I **tipi di dato** rappresentano:

- un insieme di **valori**
- un insieme di **operazioni** applicabili a questi

Ogni **tipi di dato diversi** hanno **rappresentazioni** in memoria differenti

- Il numero di celle/parole e la codifica utilizzata può cambiare

La memoria utilizzata per allocare le variabili di un determinato tipo cambia con la piattaforma (i.e., compilatore / sistema operativo / hardware)

Tipi di dato

Classificazione sulla base della struttura:

- **Tipi semplici**, informazione logicamente **indivisibile** (e.g. `int, char, float..`)
- **Tipi strutturati**: aggregazione di variabili di tipi semplici

Altra classificazione:

- **Built in**, tipi già presenti nel linguaggio base
- **User defined,** nuovi tipi creati nei programmi «componendo» variabili di tipo built in

Tipi di dato

Classificazione sulla base della struttura:

- **Tipi semplici**, informazione logicamente **indivisibile** (e.g. `int, char, float..`)
- **Tipi strutturati**: aggregazione di variabili di tipi semplici

Altra classificazione:

- **Built in**, tipi già presenti nel linguaggio base
- **User defined,** nuovi tipi creati nei programmi «componendo» variabili di tipo built in

Variabili e Tipo di Dato

In C **tutte le variabili** hanno un **tipo**, associato stabilmente mediante la **dichiarazione**

Il **tipo** di una variabile:

- definisce l'insieme dei **valori ammissibili**
- definisce l'insieme delle **operazioni applicabili**
- permette di **rilevare errori** al momento della compilazione
- definisce lo **spazio in memoria** allocato in corrispondenza alla variabile
  - Questa però dipende anche dalla piattaforma (i.e., compilatore + sistema operativo + hardware)

Tipi Semplici

```
char, int, float, double
```

Tipi Semplici

Ecco i **quattro tipi semplici** del C e la loro dimensione

- **`char:`** 1 Byte
- **`int`**: tipicamente 1 parola di memoria
- **`float:`** dipende dal compilatore (4 Byte spesso)
- **`double:`** dipende dal compilatore (più del **`float`**)

Qualificatori di tipo (per **`int`** e **`char`**)

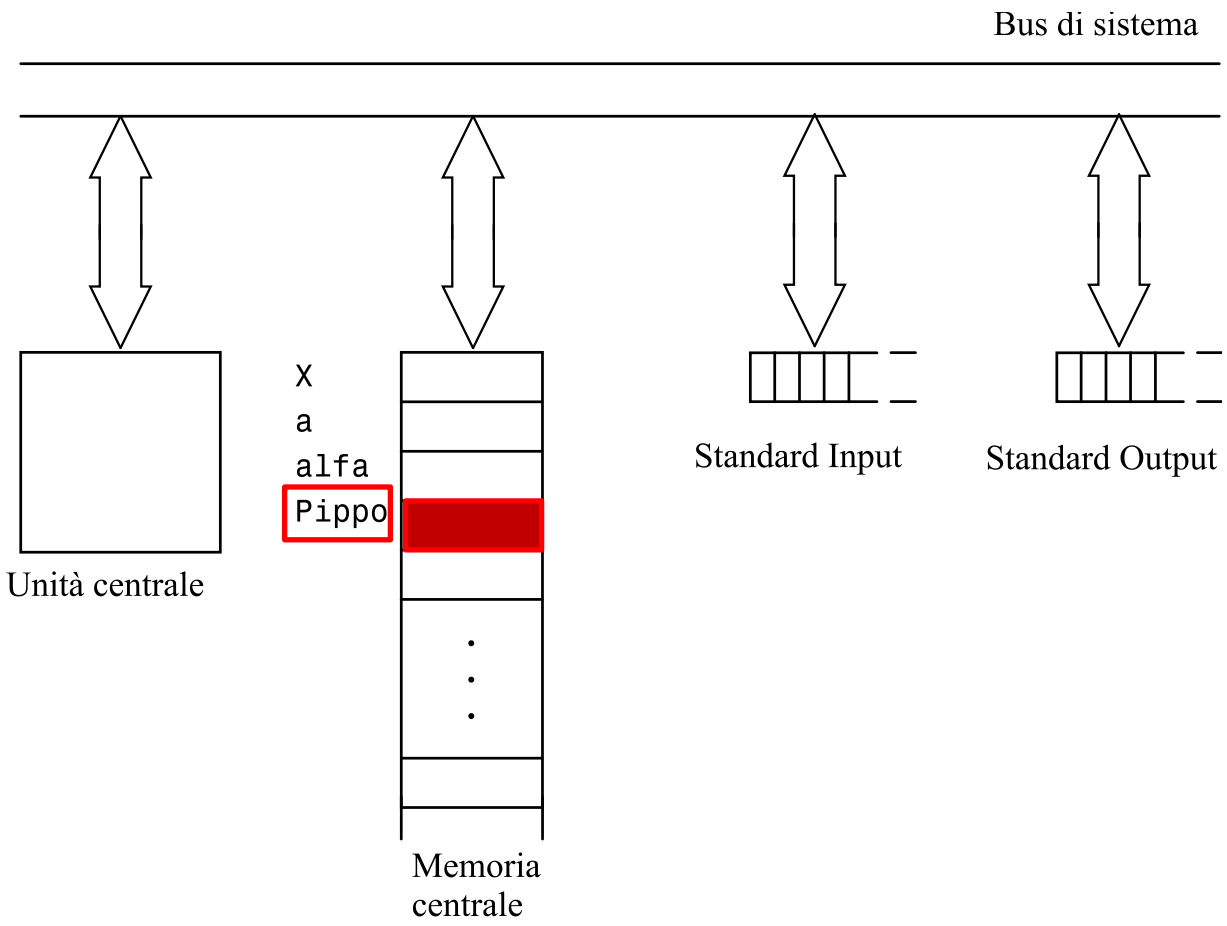
- **`signed`** utilizza una codifica con il segno (CP2)
- **`unsigned`** prevede solo valori positivi
- **NB** Allocano lo stesso spazio

Quantificatori di tipo, modificano la dimensione allocata

- **`short`** (per **`int`**)
- **`long`** (per **`int`** e **`double`**)

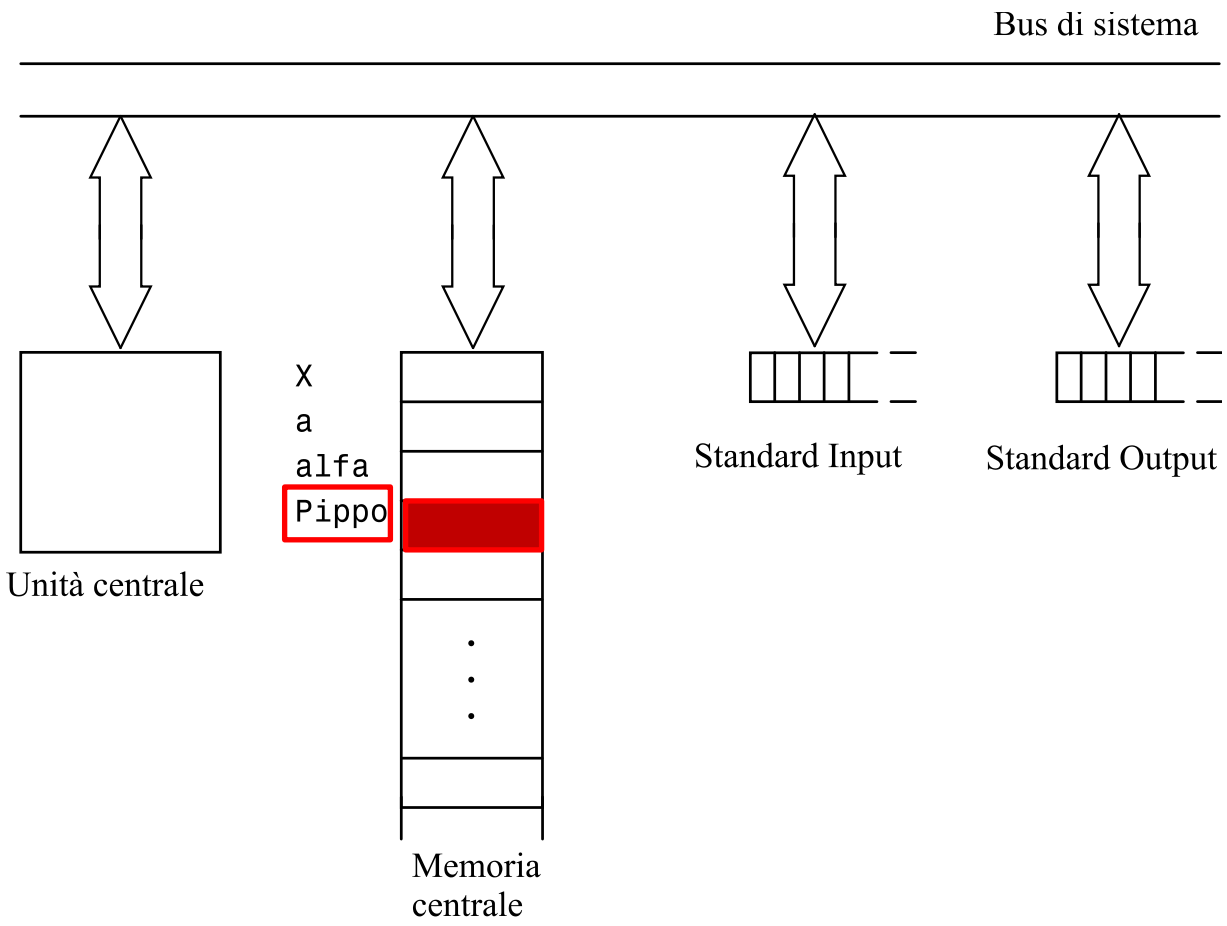
I Qualificatori, Quantificatori e lo spazio allocato

```
int Pippo; //codifica in CP2
```



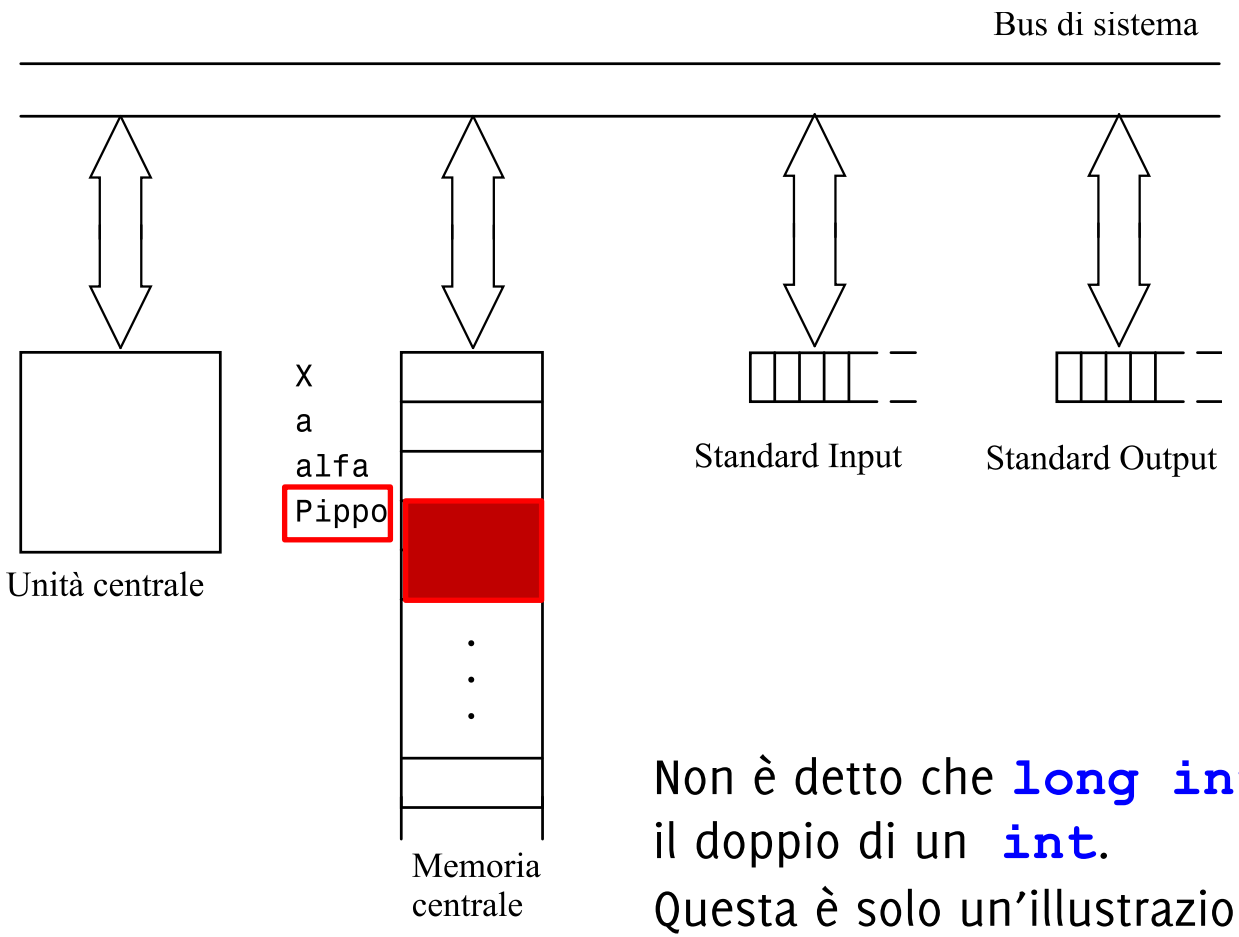
I Qualificatori, Quantificatori e lo spazio allocato

```
unsigned int Pippo; //codifica di un intero positivo
```



I Qualificatori, Quantificatori e lo spazio allocato

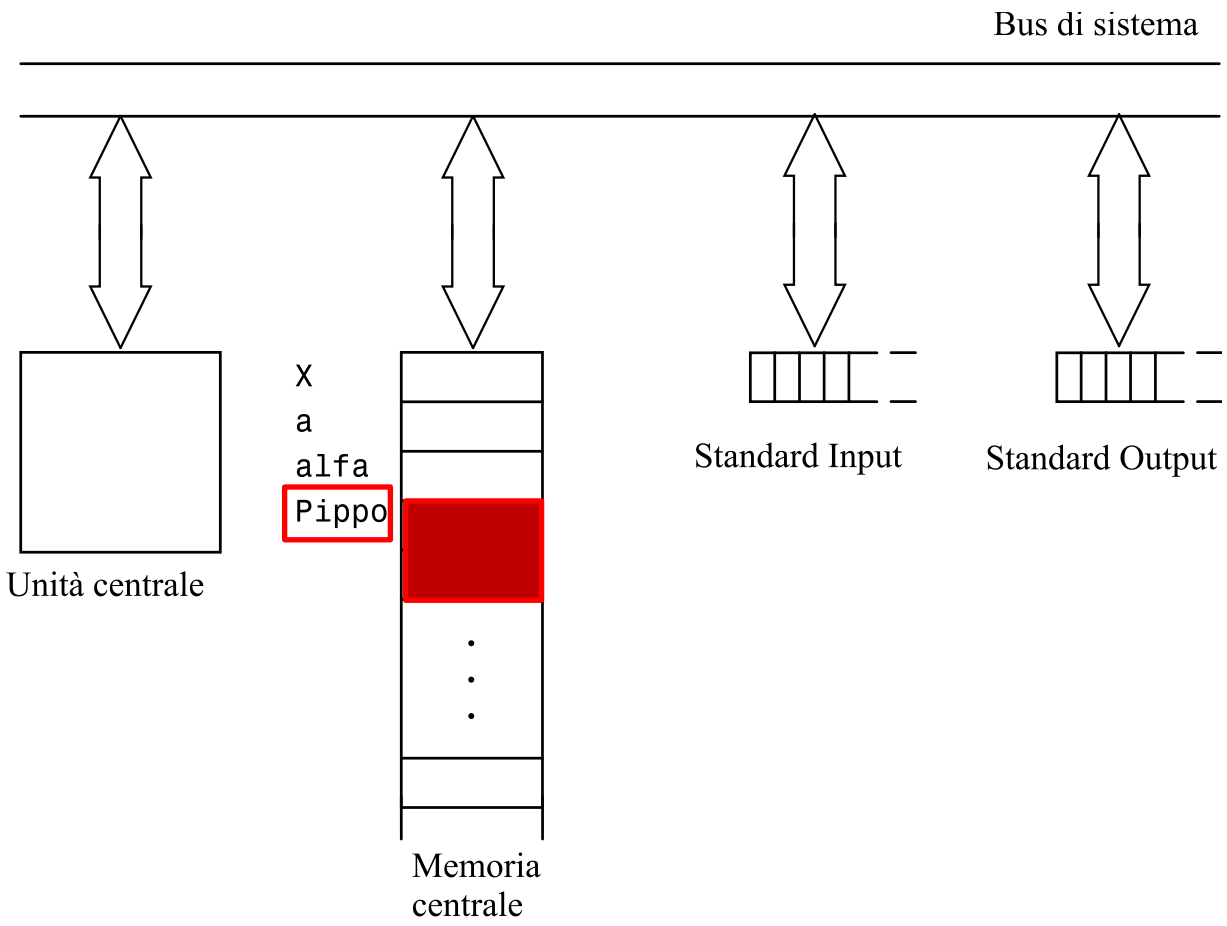
```
long int Pippo;
```



Non è detto che **`long int`** richieda il doppio di un **`int`**.
Questa è solo un'illustrazione

I Qualificatori, Quantificatori e lo spazio allocato

```
unsigned long int Pippo;
```



Il tipo `int`

Rappresentano un **sottoinsieme** di $\mathbb{N}$

Lo **spazio** allocato è tipicamente **una parola**, e dipende dalla piattaforma, oltre che dai qualificatori e quantificatori

Fatti garantiti:

- spazio (`short int`) $\leq$ spazio (`int`) $\leq$ spazio (`long int`)
- spazio (`signed int`) $=$ spazio (`unsigned int`)

Es, se la parola è a 32 bit,

- `signed int` $\{-2^{31}, \ldots, 0, \ldots +2^{31}-1\}$, i .e., 2^{32} numeri
- `unsigned int` $\{0, \ldots +2^{32}-1\}$, sempre 2^{32} numeri

Come faccio a sapere i limiti per un intero?

- `#include<limits.h>`, e richiamo le costanti `INT_MIN`, `INT_MAX`

Quando il valore di una variabile int eccede `INT_MAX` si ha overflow

Nota in C

Come faccio a scoprire quanti Byte usa il mio sistema per un intero (o un certo tipo)?

Nota in C

Come faccio a scoprire quanti Byte usa il mio sistema per un intero (o un certo tipo)?

… dichiaro un array di interi (o di un certo tipo) e faccio la differenza tra gli indirizzi di due elementi consecutivi

Nota in C

Come faccio a scoprire quanti Byte usa il mio sistema per un intero (o un certo tipo)?

… dichiaro un array di interi (o di un certo tipo) e faccio la differenza tra gli indirizzi di due elementi consecutivi

```
int main()
{
    int v[2];
    short int sv[2];
    long int lv[2];
    float f[2];
    double d[2];
    long double ld[2];
    printf("\ndimensione int: %p - %p ", &v[1], &v[0]);
    printf("\ndimensione short int: %p -  %p", &sv[1], &sv[0]);
    printf("\ndimensione long int: %p -  %p", &lv[1], &lv[0]);
    printf("\ndimensione float: %p -  %p", &f[1], &f[0]);
    printf("\ndimensione double: %p -  %p", &d[1], &d[0]);
    printf("\ndimensione long double: %p -  %p", &ld[1], &ld[0]);
    // per vedere la differenza come numero di Byte occorre il casting (che vedremo poi)

    return 0;
}
```

Nota in C

Come faccio a scoprire quanti Byte usa il mio sistema per un intero (o un certo tipo)?

… dichiaro un array di interi (o di un certo tipo) e faccio la differenza tra gli indirizzi di due elementi consecutivi

```
"C:\Users\Giacomo\Dropbox (DEIB)\Didattica\2022_Informatica_A_Boracchi\Lez7_codes\sizes.exe"

dimensione int: 000000000061FE1C - 000000000061FE18
dimensione short int: 000000000061FE16 -  000000000061FE14
dimensione long int: 000000000061FE10 -  000000000061FE0C
dimensione float: 000000000061FE08 -  000000000061FE04
dimensione double: 000000000061FDF8 -  000000000061FDF0
dimensione long double: 000000000061FDE0 -  000000000061FDD0
Process returned 0 (0x0)   execution time : 0.088 s
Press any key to continue.
```

```c
int main()
{
    int v[2];
    short int sv[2];
    long int lv[2];
    float f[2];
    double d[2];
    long double ld[2];
    printf("\ndimensione int: %p - %p ", &v[1], &v[0]);
    printf("\ndimensione short int: %p -  %p", &sv[1], &sv[0]);
    printf("\ndimensione long int: %p -  %p", &lv[1], &lv[0]);
    printf("\ndimensione float: %p -  %p", &f[1], &f[0]);
    printf("\ndimensione double: %p -  %p", &d[1], &d[0]);
    printf("\ndimensione long double: %p -  %p", &ld[1], &ld[0]);
    // per vedere la differenza come numero di Byte occorre il casting (che vedremo poi)

    return 0;
}
```

Nota in C

Le variabili `int` sono **codificate in CP2**

Se aggiungo il qualificatore `unsigned` , tutti i bit vengono usati solo per i numeri positivi. Quindi **si usa una codifica senza segno** per coprire un range maggiore.

Provare per credere…

Supponiamo di avere 32 bit per un intero. Quando dichiaro

```
int i = -1;
```

Corrisponde a scrivere -1 in una cella di memoria in CP2.

In quella cella avrò tutti i bit a 1

Però, se leggo $11\ldots11$ come `unsigned int` questo corrisponde a $2^{32}-1$

N.B `INT_MAX`, vale $2^{31}-1$ **`INT_MIN`** vale -2^{31} (perché è codificiato in CP2)

Quindi per avere $2^{32}-1$ ho **`2*INT_MAX+1`** $2(2^{31}-1)+1$

Nota in C

Le variabili **`int`** sono **codificate in CP2**

Se aggiungo il qualificatore **`unsigned`** , tutti i bit vengono usati solo per i numeri positivi. Quindi **si usa una codifica senza segno** per coprire un range maggiore.

Provare per credere…

-1 in CP2 7 bit

0000001 (+1)

1111110 (complemento)

1111111 (sommo 1)

```
2 INT_MAX   +1     1111111    (come unsigned)
              +1   0000001
2 INT_MAX + 2 (1)0000000    (come unsigned)
```

INT_MAX 0111111 (positivo in CP2)

+ 0000001

1000000 (diventa negativo in CP2)

Nota in C

Le variabili **`int`** sono **codificate in CP2**

Se aggiungo il qualificatore **`unsigned`** , tutti i bit vengono usati solo per i numeri positivi. Quindi **si usa una codifica senza segno** per coprire un range maggiore.

Provare per credere…

```
#include<limits.h>
int main()
{
    int u = -1;
    printf("\nprint as integer: %d", u);
    printf("\nprint as unsigned integer: %u", u);

    printf("\n\nprint INT_MAX as integer: %d", INT_MAX);
    printf("\nprint INT_MIN as integer: %d", INT_MIN);
    printf("\nprint INT_MAX + 1 as integer: %d", INT_MAX + 1); //OVERFLOW

    printf("\n\nprint 2 * INT_MAX + 1 as unsigned integer: %u", 2 * INT_MAX + 1);
    printf("\nprint 2 * INT_MAX +2 as unsigned integer: %u", 2 * INT_MAX + 2);
    printf("\nprint 2 * INT_MAX + 1 as integer: %d\n", 2 * INT_MAX + 1);
    return 0;}
```

`printf("%u", u);` legge il valore di u come **`unsigned int`** e lo stampa
`printf("%d", u);` legge il valore di u come **`int`** e lo stampa

Nota in C

Le variabili **`int`** sono **codificate in CP2**

Se aggiungo il qualificatore **`unsigned`** , tutti i bit vengono usati solo per i numeri positivi. Quindi **si usa una codifica senza segno** per coprire un range maggiore.

Provare per credere…

```
"C:\Users\Giacomo\Dropbox (DEIB)\Didattica\2021_Informatica_A_Boracchi\Lez7\unsign.exe"

print as integer: -1
print as unsigned integer: 4294967295

print INT_MAX as integer: 2147483647
print INT_MIN as integer: -2147483648
print INT_MAX + 1 as integer: -2147483648

print 2 * INT_MAX + 1 as unsigned integer: 4294967295
print 2 * INT_MAX + 2 as unsigned integer: 0
print 2 * INT_MAX + 1 as integer: -1

Process returned 0 (0x0)   execution time : 0.063 s
Press any key to continue.
```

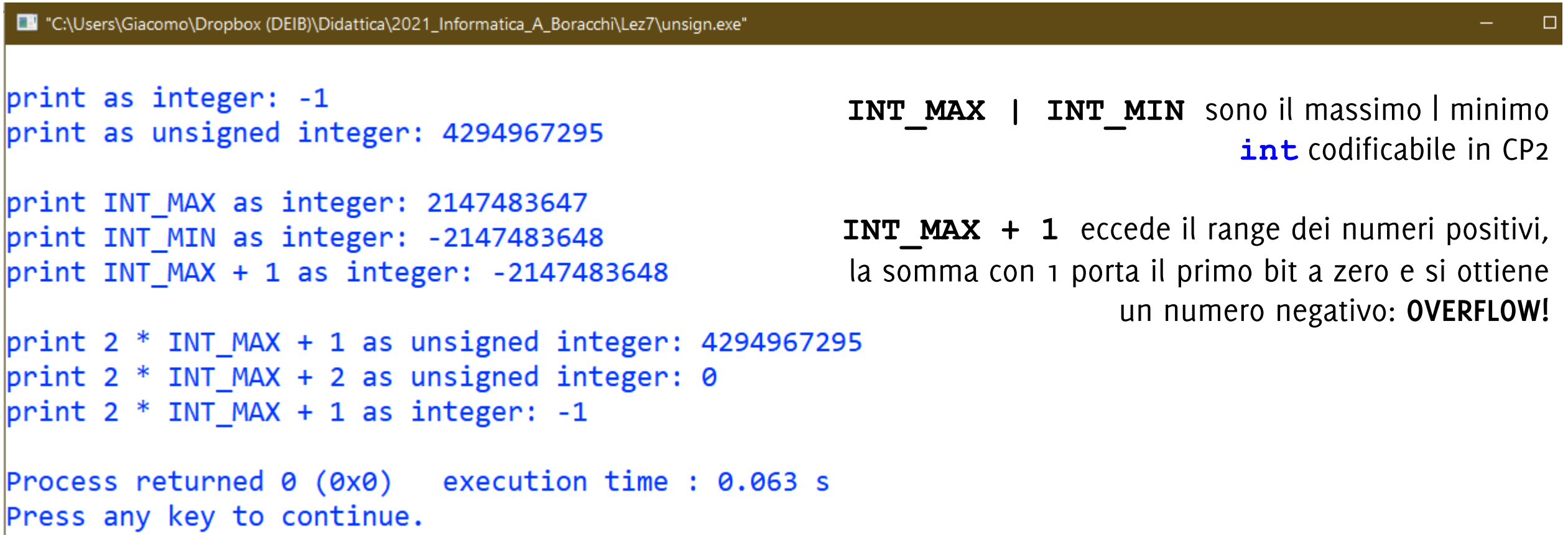
`-1` in CP2 si scrive mettendo tutti i bit disponibili a **`1`**…
Se leggo **`-1`** come **`unsigned int`** trovo il maggior numero rappresentabile

Nota in C

Le variabili **`int`** sono **codificate in CP2**

Se aggiungo il qualificatore **`unsigned`** , tutti i bit vengono usati solo per i numeri positivi. Quindi **si usa una codifica senza segno** per coprire un range maggiore.

Provare per credere…



`INT_MAX` | **`INT_MIN`** sono il massimo | minimo **`int`** codificabile in CP2

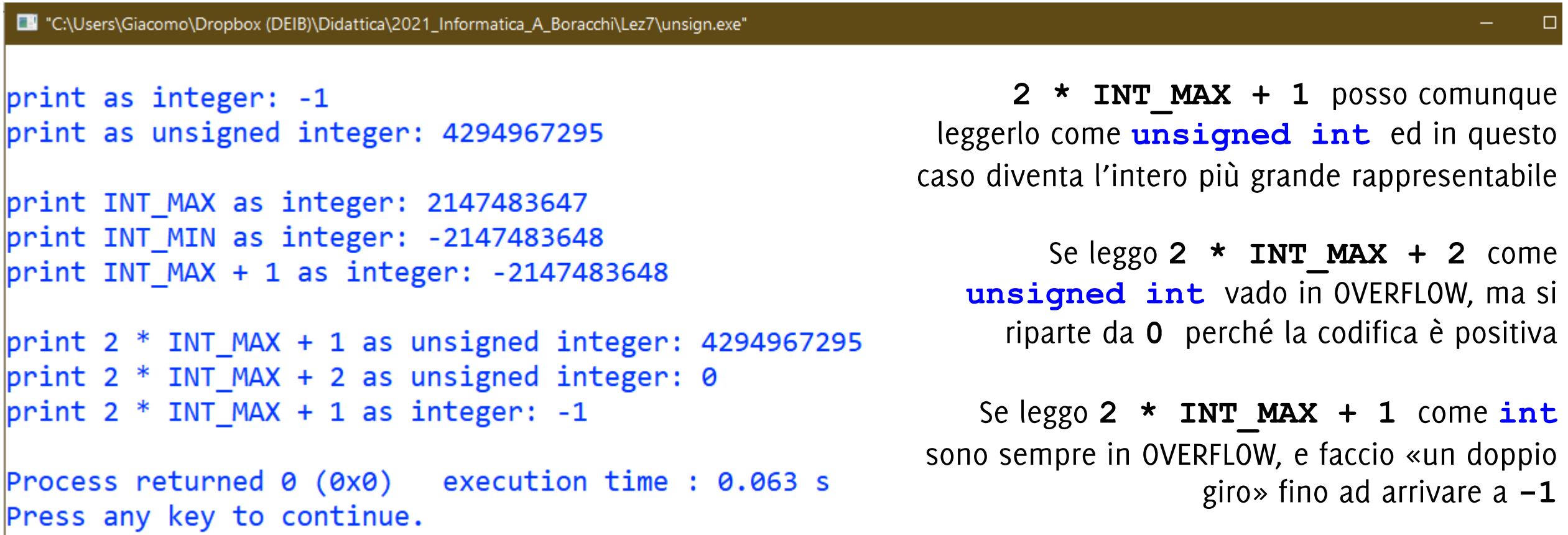
`INT_MAX + 1` eccede il range dei numeri positivi, la somma con 1 porta il primo bit a zero e si ottiene un numero negativo: **OVERFLOW!**

Nota in C

Le variabili **`int`** sono **codificate in CP2**

Se aggiungo il qualificatore **`unsigned`** , tutti i bit vengono usati solo per i numeri positivi. Quindi **si usa una codifica senza segno** per coprire un range maggiore.

Provare per credere…



`2 * INT_MAX + 1` posso comunque leggerlo come **`unsigned int`** ed in questo caso diventa l'intero più grande rappresentabile

Se leggo **`2 * INT_MAX + 2`** come **`unsigned int`** vado in OVERFLOW, ma si riparte da **`0`** perché la codifica è positiva

Se leggo **`2 * INT_MAX + 1`** come **`int`** sono sempre in OVERFLOW, e faccio «un doppio giro» fino ad arrivare a **`-1`**

Operazioni built-in 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”

Operatoori Aritmetici

Operatoori Relazionali

Il tipo `float` e `double`

Approssimazione di $\mathbb{R}$ (che è un insieme denso), quindi i **valori** vengono approssimati per «magnitudine», e limiti nella **precisione** della rappresentazione

Nella rappresentazione in virgola mobile (floating point) il numero n si scrive come due parti separate da "E":

- m: mantissa
- e: esponente (rispetto alla base 10), tali che $n = m * 10^e$

Ad esempio, 1 780 000.000 0023 in virgola mobile:

178 000.000 000 23 E1
17 800 000 000 023 E-7
1.780 000 000 0023 E+6
ecc.

Spazio su `float` e `double`

Unico fatto certo:

spazio (`float`) ≤ spazio (`double`) ≤ spazio(`long double`)

Esempio tipico `float` in 4 byte e `double` in 8 byte

⇒ accuratezza: 6 decimali per `float`

15 decimali per `double`

⇒ valori tra 10^{-38} e 10^{+38} per `float`

tra 10^{-308} e 10^{+308} per `double`

Operazioni built-in per dati di tipo `float`

=	Assegnamento di un valore float a una variabile float
+	Somma (tra float , risultato float)
-	Sottrazione (tra float , risultato float)
*	Moltiplicazione (tra float, risultato float)
/	Divisione (tra float, risultato float)
==	Relazione di uguaglianza
!=	Relazione di diversità
<	Relazione “minore di”
>	Relazione “maggiore di”
<=	Relazione “minore o uguale a”
>=	Relazione “maggiore o uguale a”

Operatoori Aritmetici

Operatoori Relazionali

Operazioni tra `float`

operazioni applicabili a **`float`** (anche a **`double`** e **`long double`**) sono le stesse degli **`int`**, ma divisione '/' dà risultato reale

NB: il simbolo dell'operazione è identico a divisione intera

standard library **`math.h`** fornisce funzioni predefinite (**`sqrt`**, **`pow`**, **`exp`**, **`sin`**, **`cos`**, **`tan`**...) applicate a valori double

$\Rightarrow$ si usa **`double`** anche quando basta **`float`**

Il tipo `float` e `double` : le approssimazioni

Nella rappresentazione di un numero decimale possono esserci **errori di approssimazione**

- Non sempre: `(x / y) * y == x`
- Per verificare l'uguaglianza tra `float` o `double`, definire dei bounds : Invece di
  - `if (x == y) ...` è meglio
  - `if (x <= y + .000001 && x >= y - .000001)`

Buona parte delle operazioni algebriche eseguibili tra `float` (es. l'elevamento a potenza, il logaritmo, la radice, il valore assoluto…) sono nella libreria math che occorre includere con

- `#include<math.h>`

Il tipo `char`

La codifica ASCII prevede di allocare sempre 1 Byte per rappresentare caratteri

- alfanumerici
- di controllo (istruzioni legate alla visualizzazione),

C'è una corrispondenza tra i **`char`** e 256 numeri interi

Le operazioni sui **`char`** sono le stesse definite su **`int`**

- hanno senso gli operatori aritmetici (+ - * / %)
- hanno senso gli operatori di relazione (== , > , < ,.. etc)

`unsigned char` coprono l'intervallo $[0, 255]$.

`signed char` coprono l'intervallo $[-128, 127]$.

N.B. non esistono tipi semplici più «piccoli» del **`char`**

Il tipo `char`

I valori costanti di tipo **`char`** nel codice sorgente si delimitano tra apici singoli `' '`

Gli apici doppi `" "` vengono utilizzati per delimitare stringhe, i.e. sequenze di caratteri (non hanno un loro tipo built in)

- le abbiamo già viste in **`printf`** e **`scanf`**

La codifica ASCII (parziale)

DEC	CAR	DEC	CAR	DEC	CAR	DEC	CAR	DEC	CAR
48	0	65	A	75	K	97	a	107	k
49	1	66	B	76	L	98	b	108	l
50	2	67	C	77	M	99	c	109	m
51	3	68	D	78	N	100	d	110	n
52	4	69	E	79	O	101	e	111	o
53	5	70	F	80	P	102	f	112	p
54	6	71	G	81	Q	103	g	113	q
55	7	72	H	82	R	104	h	114	r
56	8	73	I	83	S	105	i	115	s
57	9	74	J	84	T	106	j	116	t
				85	U			117	u
				86	V			118	v
				87	W			119	w
				88	X			120	x
				89	Y			121	y
				90	Z			122	z

Il tipo `char` esempi

```
char a,b;
b = 'q';

a = "q";

a = '\n';

b = 'ps';
a = 75;
a = 'c' + 1;
a = 'c' - 1;
  a = 20;
  a *= 4;
  a -= 10;
a = `1';
```

Il tipo `char` esempi

```
char a,b;
b = 'q'; /* Le costanti di tipo carattere si
      indicano con ' */
a = "q"; /* NO: "q" è una stringa, anche se di
      un solo carattere */
a = '\n'; /* OK: \n è un carattere a tutti gli
      effetti anche sono due elementi*/
b = 'ps'; /* NO:'ps' non è un carattere valido*/
a = 75; /*associa ad a il carattere‘K’ cfr ASCII
a = 'c' + 1; /* a diventa 'd' */
a = 'c' - 1; /* a diventa  'b' */
  a = 20;
  a *= 4; /* sta per a = a * 4, quindi a = 80 (‘P’)*/
  a -= 10; //a <--70 che corrisponde al carattere 'F'
a = `1'; /*a è il carattere 1, corrispondente a 49
```

Riepilogando sui tipi built in

I tipi **integal** sono discreti, rappresentano valori **numerabili**

- sono **`char`** ed **`int`** con tutti i qualificatori e quantificatori (**`signed/unsigned char, short/long int, signed/unsigned int`**)

I tipi **floating** approssimano insiemi **densi**

- sono **`float`** e **`double,`** eventualmente con il quantificatori **`long`**

I tipi **floating** e **integral** assieme compongono i tipi **arithmetic**

struct

Tipi strutturati

Tipi di dato

Classificazione sulla base della struttura:

- **Tipi semplici**, informazione logicamente **indivisibile** (e.g. `int, char, float..`)
- **Tipi strutturati**: aggregazione di variabili di tipi semplici

Altra classificazione:

- **Built in**, tipi già presenti nel linguaggio base
- **User defined,** nuovi tipi creati nei programmi «componendo» variabili di tipo built in

Tipi di dato

Classificazione sulla base della struttura:

- **Tipi semplici**, informazione logicamente **indivisibile** (e.g. `int, char, float..`)
- **Tipi strutturati**: aggregazione di variabili di tipi semplici
  - array
  - **struct**

Altra classificazione:

- **Built in**, tipi già presenti nel linguaggio base
- **User defined,** nuovi tipi creati nei programmi «componendo» variabili di tipo built in

Struct vs Array

Gli **array** permettono di aggregare variabili **omogenee** in una sequenza

Le `struct` permettono di aggregare variabili **eterogenee** in una sola variabile

- Le `struct` è una sorta di "contenitore" per variabili disomogenee di tipi più semplici.
- Le variabili aggregate nella struct sono dette **campi** della struct

Esempio: variabile per contenere anagrafica di impiegati

- *nome, cognome, codice fiscale, indirizzo, numero di telefono, stipendio, data di assunzione etc.*
- *Non posso metterli in un array, sono variabili diverse, è molto sconveniente metterle in variabili separate, specialmente se ho diversi impiegati*

Dichiarazione di una Struttura

Sintassi:

```
struct {
 tipo1 nomeCampo1;
 tipo2 nomeCampo2;
 …
 tipoN nomeCampoN;} nomeStruct;
```

Dichiara una variabile **struct** chiamata **nomeStruct**

I nomi dei campi della struttura sono **nomeCampo1**…

Dichiarazione compatta per campi dello stesso tipo

```
struct {
 tipo1 nomeCampoA,nomeCampoB;
…
tipoN campoN;} nomeStruct;
```

Dichiarazione di una Struttura

È possibile dichiarare due o più variabili dalla stessa struttura

```
struct {tipo1 nomeCampo1;
tipo2 nomeCampo2;
…
tipoN nomeCampoN;}nomeStruct1, nomestruct2;
```

NB: la dichiarazione di una struttura va nella **parte dichiarativa** del programma, **nel `main()`**

NB: i campi **non** sono **necessariamente** di **tipo built-in**, possono essere array o user defined (vedremo a breve)

Esempi

```
struct {
      float reale;
      float immaginaria;
      } numeroComplesso;
```

```
struct {
      int numero;
      char seme[10];
      } cartaDaGioco;
```

Esempi

```
struct {
      char Nome[30];
      char Cognome[30];
      int Stipendio;
      char CodiceFiscale[16];
      } dip1, dip2;
```

Esempi

```
struct
{
   char marca[30];
   char modello[100];
   int anno;
   int cilindrata;
   int prezzo;
} miaAuto, tuaAuto;
```

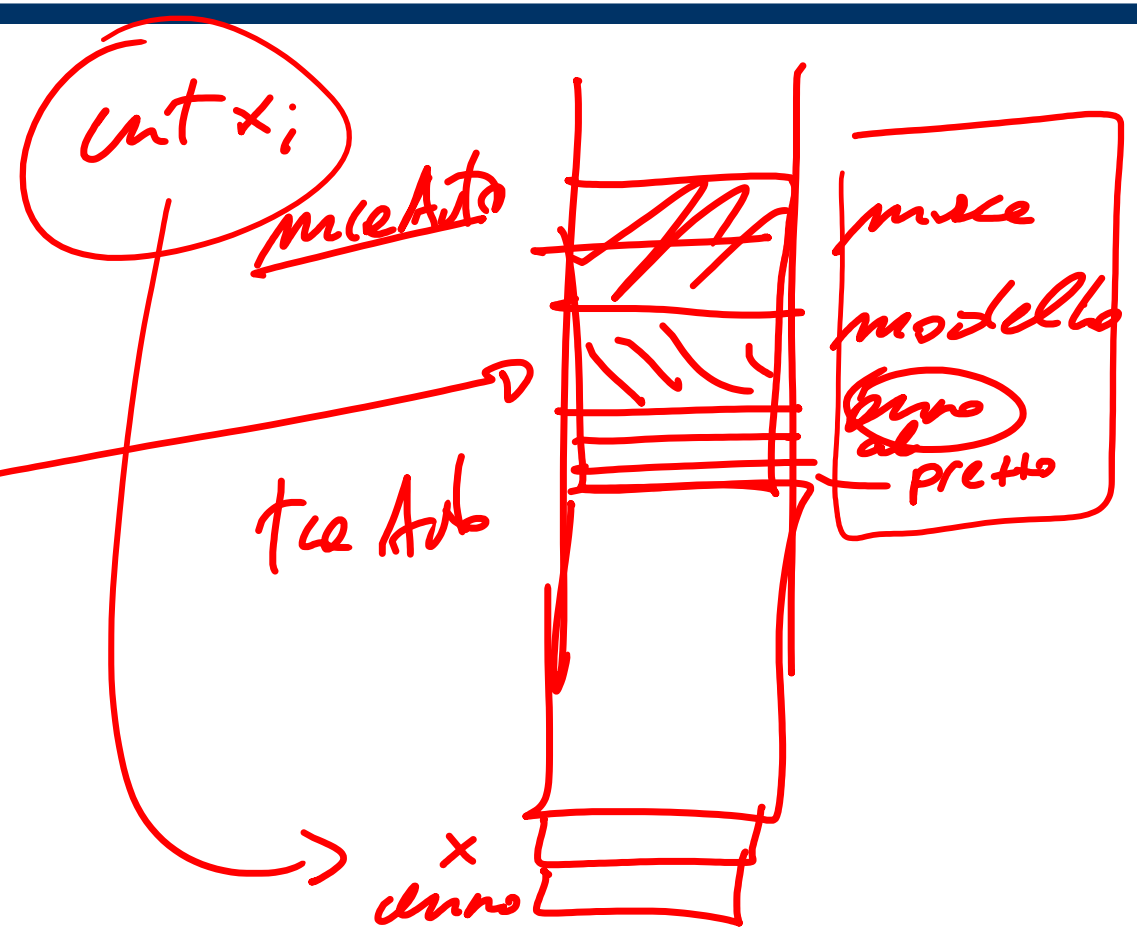
Domanda: come faccio ad assegnare nel codice un valore ai campi **marca** e **modello?**

Esempi

DICHIARAZIONE DI UNA VARIABILE

```
int main()
{
   struct
   {
      char marca[30];
      char modello[100];
      int anno;
      int cilindrata;
      int prezzo;
   } miaAuto, tuaAuto;

}
```



Domanda: come faccio ad assegnare nel codice un valore ai campi **marca** e **modello**?

Accedere ai campi di una `struct`

Per accedere ai campi si usa l'operatore ***dot*** (i.e., il punto)
Sintassi*:*

```
nomeStruct.nomeCampo;
```

Quindi, **`nomeStruct.nomeCampo`** diventa, a tutti gli effetti, **una «normale» variabile** del tipo di **`nomeCampo`**.

- **Ai campi** di una struttura applicabili tutte le **operazioni caratteristiche** del tipo di appartenenza
- In questo senso, il *dot* è "analogo" a **`[indice]`** per gli array

Accedere ai campi di una `struct`

Per accedere ai campi si usa l'operatore ***dot*** (i.e., il punto)
Sintassi:

```
nomeStruct.nomeCampo;
```

Quindi, **`nomeStruct.nomeCampo`** diventa, a tutti gli effetti, **una «normale» variabile** del tipo di **`nomeCampo`**.

- **Ai campi** di una struttura applicabili tutte le **operazioni caratteristiche** del tipo di appartenenza
- In questo senso, il *dot* è "analogo" a **`[indice]`** per gli array

Esempio

```
struct {char nome[30];
        char cognome[30];
        int stipendio;
        char codFiscale[16];
        } dip1, dip2;
// accedere ai campi di tipo semplice
dip1.stipendio = 30000;
dip2.stipendio = 2*(dip1.stipendio - 2000);
// accedere ai campi array
dip1.codiceFiscale[0] = 'K';
// copia del valore da un campo array all'altro
for(i = 0 ; i < 16 ; i++)
        dip2.codFiscale[i]=dip1.codFiscale[i];
// copia il nome di un dipendente nell'altro
strcpy(dip2.nome, dip1.nome);
dip1.cognome = dip2.cognome; // sbagliato!
```

Acquisizione e Stampa per Strutture

Non esistono caratteri speciali che permettano di usare **printf** e **scanf** direttamente su strutture.

Occorre lavorare campo per campo!

```
struct {char nome[30];
char cognome[30];
int stipendio;
} dip1;
printf("\nInserire Nome 1: ");
scanf("%s", dip1.nome); fflush(stdin);
printf("\nInserire Cognome 1: ");
scanf("%s", dip1.cognome);
printf("\nInserire Stipendio 1: ");
scanf("%d", &dip1.stipendio);
printf("%s %s, guadagna %d $",
        dip1.nome,dip1.cognome,dip1.stipendio);
```

Esempio

Definire una struttura atta a contenere una data (con mese testuale) e dichiarare due variabili **`dataNascita`** e **`dataLaurea`**.

1. Richiedere all'utente l'inserimento della data di nascita
2. Visualizzare a schermo la data di nascita
3. Definire la presunta data di laurea come
   - Giorno = giorno della nascita
   - Mese = mese della nascita
   - Anno = all'età di 24 anni
4. Stampare la presunta data di laurea

Esempio

```
#include<stdio.h>
int main()
{struct {
        int giorno;
        char mese[20];
        int anno;} N, L;
 printf("\nInserire giorno");
 scanf("%d", &N.giorno);
 printf("\nInserire mese");
 scanf("%s", N.mese);
 printf("\nInserire anno");
 scanf("%d", &N.anno);
 printf("Nato il %d  %s %d",N.giorno, N.mese, N.anno);
 L.giorno = N.giorno;
 strcpy(L.mese, N.mese);
 L.anno = N.anno + 24;
 printf("\nTi laurerai il %d  %s %d",L.giorno, L.mese, L.anno);
 return 0;}
```

Assegnamento tra Strutture

È possibile applicare **operazioni globali di assegnamento** tra **strutture identiche**.

```
struct {
    char nome[30];
    char cognome[30];
    int  stipendio;
    char codiceFiscale[16];
    } dip1, dip2;

dip1 = dip2;
```

Con l'assegnamento globale anche i valori nei campi di tipo array vengono copiati

Assegnamento tra Strutture

L'assegnamento è possibile **solo se** la **strutture sono identiche**, se cambia anche solo l'ordinamento dei campi non è possibile.

L'assegnamento globale **NON** è possibile con gli **array**

- Però, campi di strutture identiche che sono array (come nel caso di **`dip1`** e **`dip2`**) vengono assegnati correttamente!

Anche per struct, come per array, **NON** applicabili operazioni di **confronto** (**`==`**, **`!=`**)

Esempio

```
#include<stdio.h>
int main()
{struct {
        int giorno;
        char mese[20];
        int anno;} N, L;
 printf("\nInserire giorno");
 scanf("%d", &N.giorno);
 printf("\nInserire mese");
 scanf("%s", N.mese);
 printf("\nInserire anno");
 scanf("%d", &N.anno);
 printf("Nato il %d  %s %d",N.giorno, N.mese, N.anno);
 L = N;
 L.anno += 24;
 printf("\nTi laurerai il %d  %s %d",L.giorno, L.mese,
  L.anno);
 return 0;}
```

Assegnamento globale, possibile solo se **L** ed **N** sono strutture identiche.

Esempio

```
#include<stdio.h>
int main()
{struct {
        int giorno;
        char mese[20];
        int anno;} N, L;
 printf("\nInserire giorno");
 scanf("%d", &N.giorno);
 printf("\nInserire mese");
 scanf("%s", N.mese);
 printf("\nInserire anno");
 scanf("%d", &N.anno);
 printf("Nato il %d  %s %d",N.giorno, N.mese, N.anno);
 L = N;
 L.anno += 24;
 strcpy(L.mese, "dicembre\0");
 printf("\nTi laurerai il %d  %s %d",L.giorno, L.mese,
  L.anno);
 return 0;}
```

Per cambiare il mese non posso fare assegnamento tra stringhe ma devo ricorrere ad una strcpy

Le strutture devono essere perfettamente identiche per fare l'assegnamento

```
struct {
        int giorno, anno;
        char mese[20];
    } dN;

    struct {
        int giorno, anno;
        char mese[19];
    } dL;
```

…

```
dL = dN;
```

…

```
error: incompatible types when assigning
  to type 'struct <anonymous>'
 from type 'struct <anonymous>'|
```

Attenzione, il confronto tra strutture non è possibile

```c
#include<stdio.h>

int main()
{
    struct{
    float x,y;
    }p,q;

    p.x = 10.0;
    p.y = 11.0;

    q = p;

    if(p == q)
        printf("\nuguali!\n");
    else
        printf("\ndiversi!\n");

    return 0;
}
```

error: invalid operands to binary ==
(have 'struct ‹anonymous›' and 'struct ‹anonymous›')

Tipi di Dato User-Defined

Definire nuovi tipi

Tipi di dato

Classificazione sulla base della struttura:

- **Tipi semplici**, informazione logicamente **indivisibile** (e.g. `int, char, float..`)
- **Tipi strutturati**: aggregazione di variabili di tipi semplici
  - array
  - **struct**

Altra classificazione:

- **Built in**, tipi già presenti nel linguaggio base
- **User defined,** nuovi tipi creati nei programmi «componendo» variabili di tipo built in

Nuovi tipi

La keyword **typedef** permette di definire nuovi tipi in C

Sintassi:

```
typedef nomeTipo NuovoNomeTipo;
```

Es:
```
typedef int Anno;
typedef unsigned int TempAssoluta;
typedef unsigned int Eta;
```

È possibile dichiarare nuovi tipi per

- Un **tipo semplice** (**ridefinizione di tipo**)
- Un **tipo strutturato**

NB La dichiarazione di nuovi tipi va **prima** di
int main(), nel corpo del **main** potrò dichiarare variabili utilizzando
NuovoNomeTipo con la solita sintassi

Nuovi tipi

La keyword **typedef** permette di definire nuovi tipi in C

Sintassi:

```
typedef nomeTipo NuovoNomeTipo;
```

Es:
```
typedef int Anno;
typedef unsigned int TempAssoluta;
typedef unsigned int Eta;
```

È possibile dichiarare nuovi tipi per

- Un **tipo semplice** (**ridefinizione di tipo**)
- Un **tipo strutturato**

NB La dichiarazione di nuovi tipi va **prima** di
int main(), nel corpo del **main** potrò dichiarare variabili utilizzando
NuovoNomeTipo con la solita sintassi

Ridefinizione di Tipi Semplici: a che serve?

Rende più leggibile e generale il codice.

Es `typedef float MieiDati;`

Se dichiaro tutte le variabili pensate per contenere i dati di tipo **`MieiDati`** il programma è facilmente estendibile a gestire dati a precisione maggiore. Basterà sostituire

```
typedef double MieiDati;
```

Es `typedef unsigned int TempAssoluta;`

Usare **`TempAssoluta`** per dichiarare una variabile rende il codice più leggibile.

Definizione di Nuovi Tipi Strutturati

Se si combina **typedef** con un costruttore **struct** o **array** i vantaggi diventano più evidenti.

```
typedef struct {int giorno;
                char mese[20];
                int anno;} Data;
```

Quando si associa un nuovo tipo ad una struttura è possibile:

1. dichiarare **altre strutture** come variabili del nuovo tipo
2. dichiarare **array** di strutture come array del nuovo tipo
3. utilizzare il nuovo tipo come **campo** di altre **strutture**
4. utilizzare il nuovo tipo come **tipo base per nuovi tipi**

Definizione di Nuovi Tipi Strutturati

Dichiarare **altre strutture** (i.e., variabili del nuovo tipo)

```
Data oggi, domani, dopoDomani;
```

Definizione di Nuovi Tipi Strutturati

Dichiarare **array** del **nuovo tipo** (i.e., array di strutture)

```
Data calendario[365];
Data settimana[7];
Data andataRitorno[2];

// popolare andataRitorno[0] con i dati dell'andata
andataRitorno[0].giorno = 12;
strcpy(andataRitorno[0].mese, "dicembre");
andataRitorno[0].anno = 2012;

// ritorno è come l'andata
andataRitorno[1] = andataRitorno[0];

// posticipo di 10 giorni il ritorno
andataRitorno[1].giorno += 10;
```

Definizione di Nuovi Tipi Strutturati

Utilizzare il nuovo tipo come **campo** di altre **strutture**

```
struct {   char nome[30];
           char cognome[30];
           int stipendio;
           char codiceFiscale[16];
           Data dataDiNascita;} dip1;
```

Definizione di Nuovi Tipi Strutturati

Utilizzare il nuovo tipo come **tipo di campi in nuovi tipi strutturati**

```
typedef struct {char nome[30];
        char cognome[30];
        int stipendio;
        char codiceFiscale[16];
        Data dataDiNascita;
        } Dipendente;
```

Definizione di Nuovi Tipi da Array

Posso definire un nuovo tipo per variabili array

```
typedef double PioggeMensili[12];
PioggeMensili piogge87, piogge88, piogge89;
typedef double IndiciBorsa[12];
IndiciBorsa  indici87, indici88, indici89;
typedef char Stringa[12];
Stringa nome, cognome, s1;
```

È più comprensibile dell'omologo senza definizione di tipo

```
double  piogge87[12], piogge88[12], piogge89[12],
double indici87[12], indici88[12], indici89[12];
```

Definizione di Nuovi Tipi da Array

Altro esempio classico

```
typedef char Stringa[30];
```

A questo punto posso

```
typedef struct {
        Stringa nome;
        Stringa cognome;
        int stipendio;
        Stringa codFiscale;
        Data dataNascita;
        } Dipendente
```

Al posto di :

```
typedef struct {
        char nome[30];
        char cognome[30];
        int stipendio;
        char codiceFiscale[30];
        Data dataNascita;
             } Dipendente
```

È possibile dichiarare tipi used-defined a partire da altri tipi user-defined

Una Buona Regola

Utilizzare notazioni differenti per i tipi e per le variabili

Ad esempio:

- I tipi user defined iniziano con la lettera maiuscola, le variabili con la lettera minuscola

```
typedef char Stringa[30];
Stringa stringa;
```

tipo variabile

- Usare un prefisso/suffisso per i tipi, ad esempio

```
typedef char stringa_t[30];
stringa_t stringa;
```

tipo variabile

Assegnamento tra Variabili di Tipo User-Defined

Valgono le **linee guida per l'assegnamento globale per struct e per array**:

- **NON è possibile l'assegnamento** tra due variabili dello stesso tipo quando sono **array**
- **È possibile associare variabili** dello stesso tipo se queste sono di tipo **struct** (anche se contengono array nei loro campi)
- **Non** è possibile eseguire **conversioni intrinseche** tra tipi definiti dall'utente (come avviene tra i tipi built in)

Esercizio

1. Utilizzare i nuovi tipi di dato recentemente utilizzati per definire un nuovo tipo atto a descrivere un libro (con autore, titolo, costo, data di pubblicazione) ed uno scaffale di libri.
2. Scrivere un frammento di codice in cui si esegue l'acquisizione dei dati relativi ad un libro e quindi popolare uno scaffale.
3. Stampare i dati relativi a tutti i libri presenti sullo scaffale.
4. Scrivere un frammento di codice che
   a. Calcola il costo di tutti i libri presenti sullo scaffale (assumendo non vi siano più copie dello stesso libro)
   b. Copia tutti i libri con autore che ha il cognome che inizia per 'B' in una seconda variabile di tipo scaffale.

Attenzione, il confronto tra variabili di tipi strutturati non è possibile

```c
#include<stdio.h>

typedef struct
{
float x,y;
}Punto;

int main()
{
    Punto p,q;
    p.x = 10.0;
    p.y = 11.0;
    q = p;

    if(p == q)
        printf("\nuguali!\n");
    else
        printf("\ndiversi!\n");

    return 0;
}
```

error: invalid operands to binary ==
(have 'struct ‹anonymous›' and 'struct ‹anonymous›')