Programmazione

Docenti: Giovanni Da San Martino
Lamberto Ballan
<lamberto.ballan@unipd.it>

Programmazione, A.A. 2022/23
SCQ0093758 - LT Informatica

UNIVERSITÀ DEGLI STUDI DI PADOVA

Benvenuti a “programmazione 3/3”

- Benvenuti all’ultima parte di programmazione (~1/3)!
- Stessa organizzazione… e la transizione dalla prima parte dovrebbe essere “smooth”

Benvenuti a “programmazione 3/3”

UNIVERSITÀ DEGLI STUDI DI PADOVA

- Chi sono? (giusto due parole sul mio background…)

Ingegneria Informatica

PhD Ingegneria Informatica, TLC, Multimedia

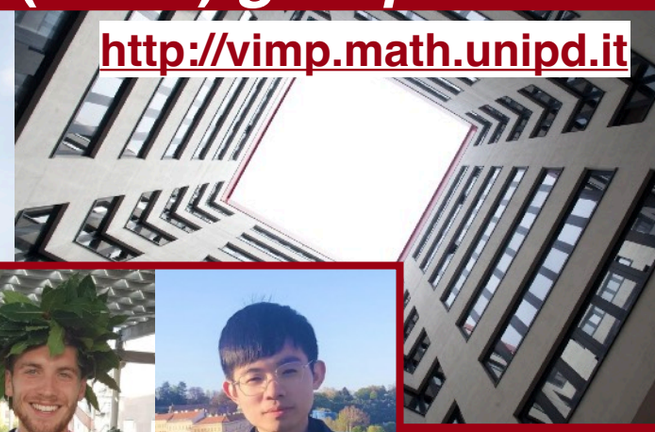
UNIVERSITÀ DEGLI STUDI DI PADOVA …

1999 — 2006 — 2011 — 2014 — 2017

UNIVERSITÀ DEGLI STUDI DI FIRENZE

Stanford University

Visual Intelligence & Machine Perception (VIMP) *group*



http://vimp.math.unipd.it

Benvenuti a “programmazione 3/3”

- Approccio algoritmico:

- “Introduzione all’Apprendimento Automatico”

  (corso opzionale al II semestre del 2 o 3 anno)

Studieremo tecniche per apprendere i programmi dai dati

Benvenuti a “programmazione 3/3”

- Argomenti rimanenti e calendario delle lezioni:
  - Definizione ed uso di strutture, unioni, enumerazioni
  - Elaborazione di File in C
  - Allocazione dinamica della memoria
  - Strutture dati elementari: liste ed alberi

Benvenuti a “programmazione 3/3”

- Argomenti rimanenti e calendario delle lezioni:
  - Definizione ed uso di strutture, unioni, enumerazioni
  - Elaborazione di File in C
  - All
  - Str

L18	W11	Monday, 8 May 2023	Lum250	*Inizio parte finale (3/3)*
L19	W11	Wednesday, 10 May 2023	Lum250	
Lab8	W11	Friday, 12 May 2023	LabP140	
L20	W12	Monday, 15 May 2023	Lum250	
L21	W12	Wednesday, 17 May 2023	Lum250	
Lab9	W12	Friday, 19 May 2023	LabP140	
L22	W13	Monday, 22 May 2023	Lum250	
L23	W13	Wednesday, 24 May 2023	Lum250	
Lab10	W13	Friday, 26 May 2023	LabP140	
	W14	Monday, 29 May 2023		*Londra (ICRA)*
E4	W14	Wednesday, 31 May 2023	Lum250	
	W14	Friday, 2 June 2023		*2 Giugno*
L24	W15	Monday, 5 June 2023	Lum250	
L25	W15	Wednesday, 7 June 2023	Lum250	
Lab11	W15	Friday, 9 June 2023	LabP140	
		Monday, 12 June 2023		*Festa Giustinianea*

Dati strutturati in C

- Le **strutture** (o “aggregati”) in C sono collezioni di variabili collegate sotto un unico nome
  - Possono contenere variabili di tipi differenti (contrariamente agli array che contengono *solo* elementi dello stesso tipo)
  - Le strutture sono comunemente usate per definire record di dati da memorizzare in file
- I puntatori e le strutture facilitano la creazione di strutture dati complesse (come as es. liste, code, pile, alberi…)

Definizione di strutture

- Le strutture sono **tipi di dati derivati**, costituite usando elementi (membri) di altri tipi
  - Una struttura viene definita con la parola chiave: **`struct`**
  - Esempio:

```
struct studente {
   char cognome[20];
   char nome[20];
   unsigned int matricola;
   unsigned int annoNascita;
   char genere;
};
```

Le variabili dichiarate entro le parentesi sono i *membri* della struttura

Tipo dei membri

- Il tipo dei membri di una `struct` può essere qualsiasi:
  - Scalare (`int`, `float`, `char`, ...)
  - Aggregato (vettori, stringhe, altre `struct`, `union`, ...)
  - Esempio:

```
struct punto {
   int x;
   int y;
};
struct rettangolo {
   struct punto altoSx;
   struct punto bassoDx;
};
```

Dichiarazione di strutture

- Le definizioni di strutture *non* riservano alcuno spazio di memoria ma creano un nuovo tipo di dati
- Le variabili di tipo struttura si dichiarano come tutte le altre variabili di tipo “standard” (i.e. int, float, char …)

```
struct studente stud, inf21_22[180], *studPtr;
```

Esempio n. 1

Dichiarazione di strutture

- Le definizioni di strutture *non* riservano alcuno spazio di memoria ma creano un nuovo tipo di dati
- Le variabili di tipo struttura si dichiarano come tutte le altre variabili di tipo “standard” (i.e. int, float, char …)

```
struct studente {
   char cognome[20];
   char nome[20];
   unsigned int matricola;
   unsigned int annoNascita;
   char genere;
} stud, inf21_22[180], *studPtr;
```

Esempio n. 2

Dichiarazione accorpata con definizione di struttura

Inizializzazione di strutture

- Le strutture possono essere inizializzate usando liste di inizializzatori (espressioni costanti), così come gli array
  - Ad esempio:

```
struct studente stud = {“Turing”,“Alan”,1001110};
```

- Cosa succede?
  - Viene creata la variabile `stud` di tipo `struct studente`
  - Si inizializzano i membri `cognome` a “Turing”, `nome` ad “Alan” e `matricola` a “1001110”
- Se nella lista vi sono *meno* inizializzatori dei membri della struttura, i restanti sono inizializzati a 0 (o NULL)

Accesso ai membri di una struttura

- Si usano due operatori per accedere ai singoli membri:
  - L'operatore di membro di struttura: **.**
  - L'operatore di puntatore a struttura: **->**
  - Vediamo due esempi:

```
printf("%s", stud.cognome);        //stampa Turing
```

(Supponendo che `studPtr` *sia stato inizializzato con l'indirizzo di* `stud`*)*

```
printf("%s", studPtr->cognome);   //stampa Turing
```

Accesso ai membri di una struttura

- Nota: l'espressioni `studPtr->cognome` è del tutto equivalente a `(*studPtr).cognome`
  - NB: l'operatore `.` ha precedenza rispetto all'operatore `*`
  - Questo significa che `*studPtr.cognome` equivarrebbe a `*(studPtr.cognome)`, ossia la variabile puntata dal membro cognome che dovrebbe essere quindi un puntatore

Operazioni su strutture

- Una variabile `struct` può essere assegnata ad un'altra (dello stesso tipo) usando l'operatore =
  - L'assegnazione avviene mediante copia del contenuto (i.e. non del puntatore)
  - Questo vale anche per le stringhe (se uno dei membri della struttura è una stringa, viene copiata l'intera stringa e non il suo puntatore), per gli array, …

Strutture e `typedef`

- E’ buona regola combinare la definizione di strutture con `typedef` per maggiore leggibilità
  - Ad esempio:

```
struct studente {
   char cognome[20];
   char nome[20];
   unsigned int matricola;
   unsigned int annoNascita;
   char genere;
};
typedef struct studente Studente;
Studente stud;
```

Strutture e `typedef`

- In sostanza `typedef` viene usato per assegnare nomi simbolici (alias) a tipi di dato
  - Solitamente `struct` e `typedef` vengono usati assieme, ma un esempio classico di utilizzo di `typedef` è anche il seguente:

```
typedef unsigned char Byte;
```

- Vi è una certa somiglianza tra `typedef` e `#define` (entrambi permettono di definire alias / nomi simbolici), tuttavia:
  - `typedef` è limitato alla definizione di alias per tipi
  - `#define` è "più generale" (ad es. possiamo definire alias per valori) ed è processato dal pre-processore, mentre le istruzioni `typedef` sono processate dal compilatore

Allocazione dei membri di `struct`

- I membri di una struttura sono allocati contiguamente in memoria, seguendo lo stesso ordine di definizione
  - Tra un membro ed il successivo possono esserci spazi intermedi di "allineamento" della memoria
    - Tali spazi non sono indirizzabili e hanno contenuto indefinito
    - La presenza di spazi intermedi dipende dal microprocessore (molti richiedono che le variabili siano allocate a indirizzi multipli di un certo valore, ad es 4 byte, 8 byte, …)

Operatore `sizeof`

- In tipi possono occupare un numero diverso di byte a seconda dell'architettura / compilatore
  - Ad esempio `int` può occupare 2 o 4 bytes
- Un operatore molto utile (ed usato) in C è `sizeof`
  - Si tratta di un operatore unario che restituisce il numero di bytes necessario ad allocare un tipo o una variabile

```
sizeof(int);   // 4
```

```
char nome[20];
sizeof(nome);  // 20
```

Strutture e funzioni

- Essendo a tutti gli effetti nuovi tipi, possiamo usare le strutture come parametri di funzione
  - Perciò il passaggio delle variabili avviene per valore (a meno che non si effettui il passaggio di un puntatore)
  - Una funzione può anche restituire una variabile di tipo `struct`

- Nota: se i parametri / valore restituito sono `struct`, può esserci un overhead significativo a run time

Strutture e funzioni

- Perché chiamante e funzione chiamata "riconoscano" la stessa `struct`, la definizione deve esse unica
  - Nel caso (frequente) di progetti con file multipli, la definizione della struttura deve essere ripetuta in tutti i file
    - Tipicamente questa viene inserita in un file di header
  - Più precisamente le definizioni della struttura devono essere tra loro *compatibili*
    - Ciò significa che devono avere stesso numero di membri, con lo stesso nome, nello stesso ordine e con tipi tra loro compatibili

Strutture *vs* unioni

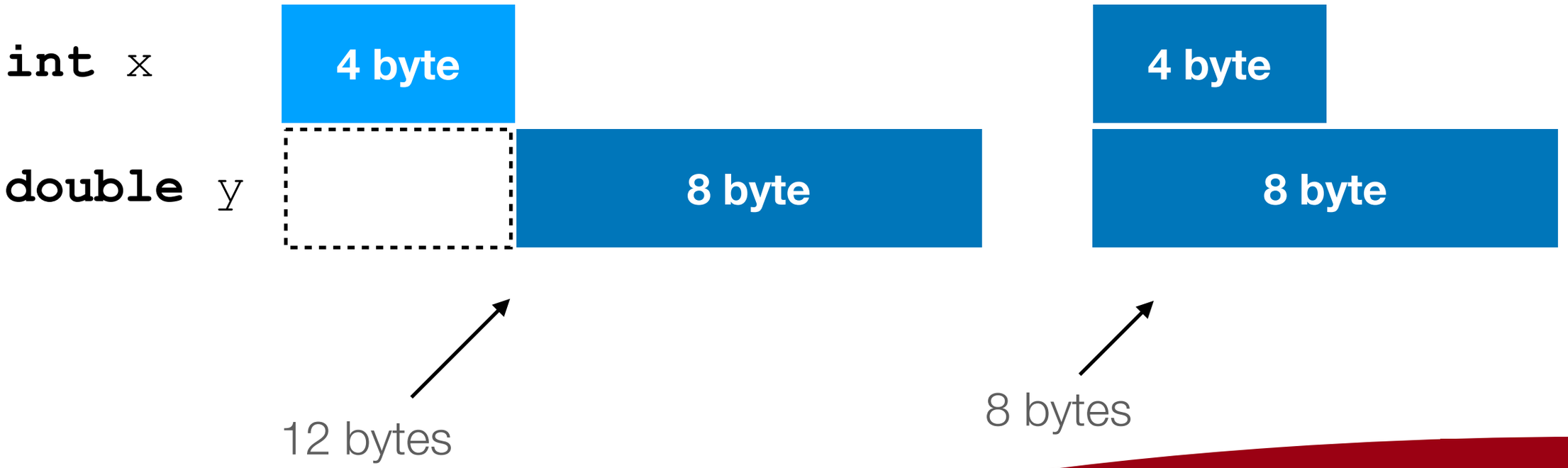
- Come le strutture, un'unione è un tipo di dati derivato
  - Una unione viene definita con la parola chiave: **`union`**
  - Contrariamente a quanto avviene nelle strutture, tutti i membri di una unione condividono la stessa memoria
  - I membri di una unione possono essere di qualunque tipo, ed il numero di byte utilizzati per memorizzare una unione è pari a quello dell'elemento più grande
- Strutture e unioni sono quindi molto simili tra loro, ma le unioni hanno lo scopo di evitare l'eventuale spreco di memoria causato da membri non utilizzati

Strutture *vs* unioni

- Vediamo un esempio:

```
struct numero {
   int x;
   double y;
};
```

```
union numero {
   int x;
   double y;
};
```



Campi di bit

- Il C consente di specificare il numero di bit con cui memorizzare un membro `int` o `unsigned int`
  - Questo costrutto viene chiamato campo di bit
  - Può essere definito sia per strutture che per unioni
  - L'obiettivo dei campi di bit è quello di ottimizzare / minimizzare il numero di bit necessari per rappresentare il dato

Campi di bit

- I campi di bit si specificano all'interno della definizione della struttura col simbolo `:` seguito da `<num_bit>`
  - Vediamo un esempio:

```
struct cartaBit {
   unsigned int valore: 4;
   unsigned int seme: 2;
   unsigned int colore: 1;
};
```

- La costante intera `<num_bit>` rappresenta la larghezza del campo (ed il numero di bit si basa sull'intervallo di valori voluto)
- Avremo quindi un numero tra 0 e il numero max di bit necessari per memorizzare un int (ad es. 32, ma può variare a seconda dell'architettura specifica)

Campi di bit

- I campi di bit sono accorpati in modo da costituire gruppi di byte chiami *storage unit*
  - Ciascuna storage unit è pari alla dimensione di un int
  - L'ordine di memorizzazione dei campi nelle storage unit dipende dal particolare compilatore / implementazione
  - In generale non è quindi possibile determinare con esattezza la posizione di un campo di bit: pertanto non è possibile definire un puntatore diretto a campo di bit
  - Si possono definire campi anonimi (senza assegnare un nome al campo) che possono essere utilizzati per "riempire" i bit di una storage unit e forzare la memorizzazione del membro seguente all'inizio della successiva storage unit

Campi di bit

- Vediamo un esempio con campo di bit anonimo:

```
struct cartaBit {
  unsigned int valore: 4;
  unsigned int seme: 2;
  unsigned int   : 26;
  unsigned int colore: 1;
};
```

- D: quanti byte occupa una variabile cartaBit? R: **8**

- Nota: si può forzare l'allineamento al prossimo membro così:

```
  unsigned int   : 0;
```

Contatti

- **Ufficio:** Torre Archimede, ufficio 6CD3
- **Ricevimento:** Venerdì 8:30-10:30 *(e-mail per conferma)*

✉ lamberto.ballan@unipd.it
🏠 http://www.lambertoballan.net
🏠 http://vimp.math.unipd.it
@ twitter.com/lambertoballan