

**Struttura**

# Strutture

- Collezioni di variabili collegate sotto un unico nome
- Possono contenere variabili di tipi di dati differenti
- Diverse dagli array che contengono solo elementi dello stesso tipo di dati

# Strutture

- Le strutture sono tipo derivati costruite usando oggetti di altri tipi
- La parola chiave `struct` introduce la definizione di una struttura

```
struct card {  
    const char *face;  
    const char *suit;  
};
```

- L'identificatore `card` è l'etichetta della struttura
- Le variabili dichiarate dentro le parentesi di `struct` sono i membri della struttura
- I membri di `struct` devono avere nomi unici
- Tipi di strutture separate possono contenere membri dello stesso nome

# Strutture

- I membri di una struttura possono essere
  - variabili di tipi di dati primitivi (es. `int`, `double`)
  - Aggregati come array o altri oggetti di tipo `struct`
- Esempio di struttura con membri di tipi differenti

```
struct employee {  
    char firstName[20];  
    char lastName[20];  
    int age;  
    double hourlySalary;  
};
```

# Strutture autoreferenziali

- Un tipo `struct` non può contenere una variabile di quello stesso tipo di `struct`
- Un tipo di `struct` può contenere un **puntatore** a quel tipo di `struct`

```
struct employee {  
    char firstName[20];  
    char lastName[20];  
    unsigned int age;  
    double hourlySalary;  
    struct employee *managerPtr; // pointer  
};
```

- Una struttura contenente un membro che è un puntatore allo stesso tipo di struttura è detta **struttura autoreferenziale**
- Le strutture autoreferenziali sono usate per costruire strutture di dati collegate

# Definizione di variabili di strutture

- Le definizioni di strutture non riservano alcuno spazio in memoria
- Ogni definizione crea un nuovo tipo di dato usato per definire le variabili
- Le istruzioni seguenti riservano memoria per le variabili usando il tipo `struct card`

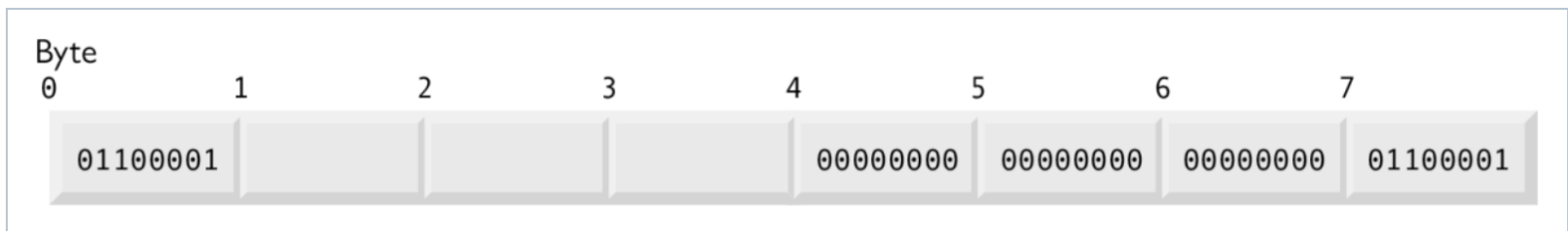
```
struct card myCard;  
struct card deck[52];  
struct card *cardPtr;
```

- Le variabili di un dato tipo di struttura possono anche essere definite nella definizione della struttura

```
struct card {  
    const char *face;  
    const char *suit;  
} myCard, deck[52], *cardPtr;
```

# Confrontare oggetti struttura

- Confrontare oggetti struttura non è consentito!
- Le strutture non possono essere confrontate usando gli operatori `==` e `!=` perché i membri della struttura potrebbero non essere memorizzati in byte di memoria consecutivi
- ```
struct example {  
    char c;  
    int i;  
} sample1, sample2;
```
- Un computer con parole di quattro byte può richiedere che ogni membro di `struct example` sia allineato su un confine di parola
- Esempio con `c='a'` e `i=97`



# Inizializzazione di strutture

- Come per gli array è possibile inizializzare variabili di tipo `struct` con una lista di inizializzatori

```
struct card myCard = {"Three", "Hearts"};
```

- inizializza il membro `face` a «Three» e il membro `suit` a «Hearts»
- Se vi sono meno inizializzatori che membri, i restanti membri sono inizializzati a 0 o NULL per i membri puntatore
- Le variabili di tipo struttura definite fuori da una definizione di funzione sono inizializzati a 0 o NULL e non sono esplicitamente inizializzati nella definizione esterna
- Si posso assegnare variabili struttura ad altre variabili di struttura dello stesso tipo
- Si posso assegnare valori a singoli membri della struttura



# Accesso ai membri delle strutture

- E' possibile accedere ai membri di una struttura con:
  - **L'operatore di membro di struttura** (.), o  
operatore punto
  - **L'operatore di puntatore a struttura** (->), o  
operatore freccia

# Operatore di membro di struttura

- L'operatore di membro di struttura accede a un membro di una struttura tramite il nome di una variabile di struttura
- Possiamo stampare il membro `suit` con l'istruzione

```
printf("%s", myCard.suit);
```

# Operatore di puntatore a struttura

- Accesso ad un membro di una struttura tramite un puntatore a struttura

```
printf("%s", cardPtr->suit);
```

- L'espressione `cardPtr->suit` è equivalente a `(*cardPtr).suit`
- Le parentesi sono necessarie perché l'operatore di membro di struttura (`.`) ha una precedenza più alta dell'operatore che dereferenzia il puntatore (`*`)

# Esempio

```
1 // fig10_01.c
2 // Operatore di membro di struttura e
3 // operatore di puntatore a struttura
4 #include <stdio.h>
5
6 // definizione della struttura card
7 struct card {
8     const char *face; // definisci il puntatore face
9     const char *suit; // definisci il puntatore suit
10 };
11
12 int main(void) {
13     struct card myCard; // definisci una variabile di tipo struct card
14
15     // inserisci le stringhe in myCard
16     myCard.face = "Ace" ;
17     myCard.suit = "Spades" ;
18
19     struct card *cardPtr = &myCard; // assegna l'indirizzo di myCard a cardPtr
20
21     printf("%s of %s\n", myCard.face, myCard.suit);
22     printf("%s of %s\n", cardPtr->face, cardPtr->suit);
23     printf("%s of %s\n", (*cardPtr).face, (*cardPtr).suit);
24 }
```

Ace of Spades  
Ace of Spades  
Ace of Spades

# Uso delle strutture con le funzioni

- I singoli membri di una struttura e gli interi oggetti struttura sono **passati per valore**
  - Le funzioni non possono modificarli nella funzione chiamante
  - Per passare una struttura per riferimento, usate l'indirizzo dell'oggetto struttura
  - Gli array di oggetti struttura sono passati per riferimento (come tutti gli array)
- Per passare un array per valore utilizzate una struttura
  - Create una struttura con un array come membro
  - Le strutture sono passate per valore, quindi i suoi membri sono passati per valore



# **`typedef` per nomi più «umani»**

- `typedef` consente di creare sinonimi (*alias*) per tipi di dati precedentemente definiti
- Viene usata per creare nomi più corti per i tipi di strutture e semplificare le dichiarazioni di tipi come i puntatori a funzioni

```
typedef struct card Card;
```

- Potete usare `Card` per dichiarare variabili di tipo `struct card`

```
Card deck[52];
```

- `typedef` non crea un nuovo tipo, ma solo un nome alternativo

# **typedef e definizioni di struct**

- `typedef` può essere usato per definire un tipo di struttura
- Non è necessaria l'etichetta della struttura

```
typedef struct {  
    const char *face;  
    const char *suit;  
} Card;
```

# Caso pratico di simulazione di numeri casuali: mescolamento e distribuzione di carte ad alte prestazioni

- Deitel code example `fig10_02.c`
- L'algoritmo seleziona un numero casuale tra 0 e 51 per ogni carta, quindi scambia la carta attuale con quella scelta
- Viene eseguito un totale di 52 scambi durante un'unica passata attraverso l'intero array, mescolando così le carte
- Poiché le carte sono state scambiate all'interno dell'array, l'algoritmo altamente performante per la distribuzione (funzione `deal`, righe 62-68) può distribuire le carte mescolate con una sola passata dell'array



# Intervallo



Fonte: chatGPT

# Operatori bit a bit

- I computer rappresentano tutti i dati internamente come sequenze di bit
- Ogni bit può assumere valore 0 o 1
- Una sequenza di 8 bit forma un byte
- Gli operatori bit a bit sono usati per manipolare i bit di operandi interi
  - Sia interi signed che unsigned, di solito vengono usati gli interi senza segno

# Operatori bit a bit

| Operatore                                                                    | Descrizione                                                                                                                                                                                          |
|------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>&amp;    AND bit a bit</b>                                                | Confronta i suoi due operandi bit per bit. I bit nel risultato sono impostati a <b>1</b> se i bit corrispondenti nei due operandi sono <i>entrambi</i> <b>1</b> .                                    |
| <b>     OR inclusivo bit a bit</b>                                           | Confronta i suoi due operandi bit per bit. I bit nel risultato sono impostati a <b>1</b> se <i>almeno uno</i> dei bit corrispondenti nei due operandi è <b>1</b> .                                   |
| <b>^    OR esclusivo bit a bit<br/>(conosciuto anche come XOR bit a bit)</b> | Confronta i suoi due operandi bit per bit. I bit nel risultato sono impostati a <b>1</b> se i bit corrispondenti nei due operandi sono differenti.                                                   |
| <b>&lt;&lt;   spostamento a sinistra</b>                                     | Sposta a sinistra i bit del primo operando del numero di bit specificato dal secondo operando; i bit da destra vengono riempiti con <b>0</b> .                                                       |
| <b>&gt;&gt;   spostamento a destra</b>                                       | Sposta a destra i bit del primo operando del numero di bit specificato dal secondo operando; il metodo di riempimento da sinistra è dipendente dalla macchina quando l'operando sinistro è negativo. |
| <b>~    complemento</b>                                                      | Tutti i bit <b>0</b> sono impostati a <b>1</b> e tutti i bit <b>1</b> sono impostati a <b>0</b> ("alternanza di bit").                                                                               |

# Stampa della rappresentazione a bit di un intero

```
1 // fig10_04.c
2 // Stampa di un int senza segno nella sua rappresentazione in bit
3 #include <stdio.h>
4
5 void displayBits(unsigned int value); // prototipo
6
7 int main(void) {
8     unsigned int x = 0; // variabile per memorizzare l'input dell'utente
9
10    printf("%s", "Enter a nonnegative int: ");
11    scanf("%u", &x);
12    displayBits(x);
13 }
14
15 // stampa i bit di un valore int senza segno
16 void displayBits(unsigned int value) {
17     // definisci displayMask ed effettua uno spostamento a sinistra di 31 bit
18     unsigned int displayMask = 1 << 31;
19
20    printf("%10u = ", value);
21
22    // effettua un'iterazione attraverso tutti i bit
23    for (unsigned int c = 1; c <= 32; ++c) {
24        putchar(value & displayMask ? '1' : '0');
25        value <<= 1; // sposta value a sinistra di 1 bit
26
27        if (c % 8 == 0) { // stampa uno spazio dopo 8 bit
28            putchar(' ');
29        }
30    }
31
32    putchar('\n');
33 }
```

```
Enter a nonnegative int: 65000
65000 = 00000000 00000000 11111101 11101000
```

# Stampa della rappresentazione a bit di un intero

- La funzione displayBits utilizza l'operatore bit a bit AND per combinare la variabile value con la variabile displayMask
- Spesso l'operatore bit a bit AND è usato con un operando chiamato **maschera**, un valore intero con bit specifica impostati a 1
- Le maschere servono per nascondere alcuni bit in un valore mentre vengono selezionati altri bit

# Uso degli operatori bit a bit di spostamento a sinistra e a destra

- L'operatore di spostamento a sinistra (<<)
  - sposta a sinistra i bit del suo operando sinistro del numero di bit specificato nel suo operando destro
  - I bit vuoti. Destra sono sostituiti con zeri
  - I bit spostati a sinistra vanno persi
- L'operatore di spostamento a destra (>>) sposta a destra i bit del suo operando sinistro del numero di bit specificato nel suo operando destro
  - Lo spostamento a destra di un unsigned int fa sì che vengano sostituiti con zeri tutti i bit rimasti vuoti a sinistra
  - I bit spostati a destra vanno perduti

# Uso degli operatori bit a bit di spostamento a sinistra e a destra

```
1 // fig10_06.c
2 // Uso degli operatori di spostamento bit a bit
3 #include <stdio.h>
4
4 void displayBits(unsigned int value); // prototipo
5
5 int main(void) {
6     unsigned int number1 = 960; // inizializza number1
6
7     // illustra lo spostamento a sinistra bit a bit
8     puts("\nThe result of left shifting");
9     displayBits(number1);
10    puts("8 bit positions using the left shift operator << is");
11    displayBits(number1 << 8);
11
12    // illustra lo spostamento a destra bit a bit
13    puts("\nThe result of right shifting");
14    displayBits(number1);
15    puts("8 bit positions using the right shift operator >> is");
16    displayBits(number1 >> 8);
17 }
```

# Uso degli operatori bit a bit di spostamento a sinistra e a destra

```
18 // stampa i bit di un valore unsigned int
19 void displayBits(unsigned int value) {
20     // dichiara displayMask ed effettua uno spostamento a sinistra di 31 bit
21     unsigned int displayMask = 1 << 31;
22
23     printf("%10u = ", value);
24
25     // effettua un'iterazione attraverso tutti i bit
26     for (unsigned int c = 1; c <= 32; ++c) {
27         putchar(value & displayMask ? '1' : '0');
28         value <<= 1; // sposta value a sinistra di 1 bit
29
30         if (c % 8 == 0) { // stampa uno spazio dopo 8 bit
31             putchar(' ');
32         }
33     }
34
35     putchar('\n');
```

The result of left shifting

960 = 00000000 00000000 00000011 11000000

8 bit positions using the left shift operator << is

245760 = 00000000 00000011 11000000 00000000

The result of right shifting

960 = 00000000 00000000 00000011 11000000

8 bit positions using the right shift operator >> is

3 = 00000000 00000000 00000000 00000011



# assegnazione bit a bit

- Ogni operatore bit a bit ha un operatore di assegnazione corrispondente

---

## Operatori di assegnazione bit a bit

---

---

**&=** Operatore di assegnazione AND bit a bit.

---

**|=** Operatore di assegnazione OR inclusivo bit a bit.

---

**^=** Operatore di assegnazione OR esclusivo bit a bit.

---

**<<=** Operatore di assegnazione di spostamento a sinistra.

---

**>>=** Operatore di assegnazione di spostamento a destra.

---

# Costanti di enumerazione

- Enum definisce un insieme di **costanti di enumerazione** intere rappresentate da identificatori
- I valori in un enum partono da 0, a meno che non sia specificato diversamente, e sono incrementate di 1

```
enum months {  
    JAN, FEB, MAR, APR, MAY, JUN,  
    JUL, AUG, SEP, OCT, NOV, DEC};
```

- Crea un nuovo tipo enum months, in cui gli edintificatori sono impostati agli interi da 0 a 11

# Costanti di enumerazione

```
enum months {  
    JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG,  
    SEP, OCT, NOV, DEC};
```

- Enumera i mesi da 1 a 12
- Gli identificatori in qualsiasi enumerazione devono essere unici
- Il valore di ogni costante di enumerazione può essere impostato esplicitamente nella definizione assegnando un valore all'identificatore
- Più membri di un'enumerazione possono avere lo stesso valore
- Usare lettere maiuscole nei nomi delle costanti per ricordarvi che sono costanti e non variabili

# Costanti di enumerazione

```
1 // fig10_08.c
2 // Uso di un'enumerazione
3 #include <stdio.h>
4
5 // le costanti di enumerazione rappresentano i mesi dell'anno
6 enum months {
7     JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
8 };
9
10 int main(void) {
11     // inizializza un array di puntatori
12     const char *monthName[] = { "", "January", "February", "March",
13     "April", "May", "June", "July", "August", "September", "October",
14     "November", "December" };
15
16     // effettua un'iterazione attraverso months
17     for (enum months month = JAN; month <= DEC; ++month) {
18         printf("%2d%11s\n", month, monthName[month]);
19     }
20 }
```

```
1 January
2 February
3 March
4 April
5 May
6 June
7 July
8 August
9 September
10 October
11 November
12 December
```



# Recap

- Strutture (`struct`) per contenere variabili di tipi differente
- Istruizione `typedef`
- Operatori bit a bit
- Costanti di enumerazione