Capitolo 16

Strutture, unioni ed enumerazioni

Variabili strutturate (strutture)

- Una ***struttura*** è una variabile composta da vari elementi
- È diversa da un array:
  - Gli elementi di una struttura - i suoi ***membri*** - non devono avere tutti lo stesso tipo
  - Ogni membro ha un nome (non un indice)
- In alcuni linguaggi le strutture sono dette ***record,*** ed i membri sono i ***campi*** del record.

Dichiarazione di strutture

- Una struttura serve a memorizzare informazioni collegate fra loro
- Definisce un "tipo"
- Ecco un esempio di dichiarazione di due variabili struttura che contengono informazioni su oggetti in un magazzino:

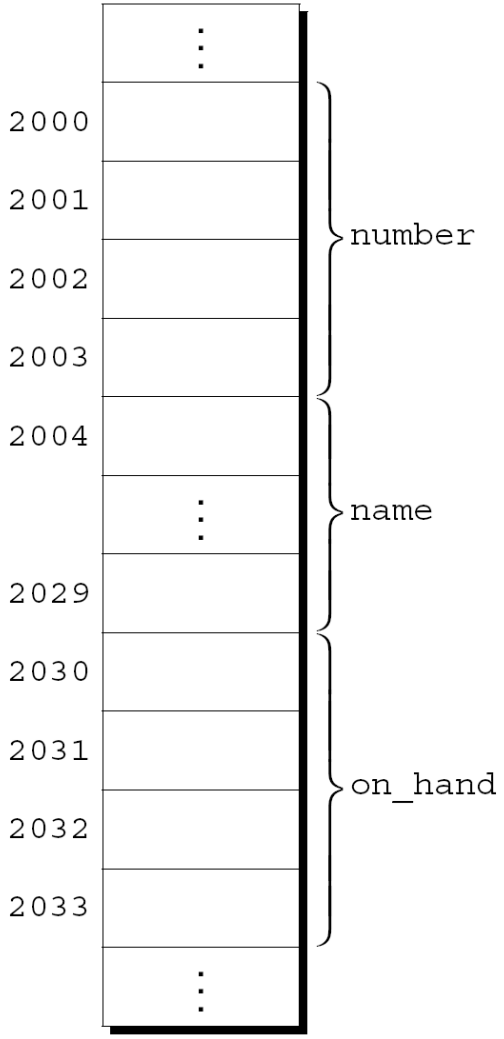
```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;
```

Tipo

Variabili

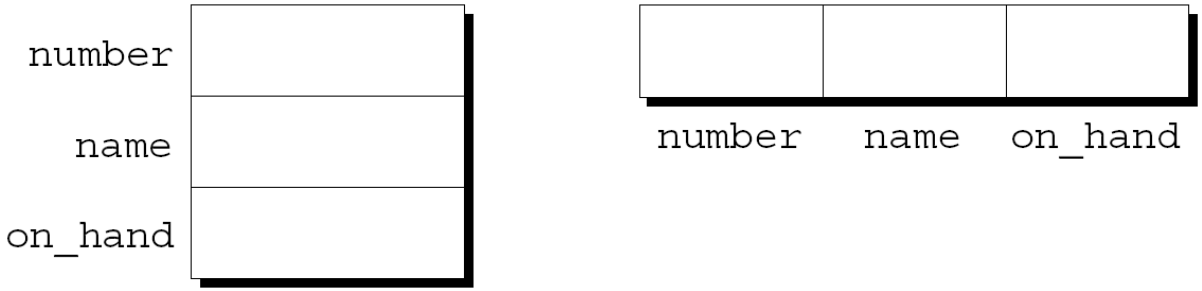
Dichiarazione di strutture

- I membri di una struttura vengono memorizzati nell'ordine in cui appaiono nella dichiarazione
- Memoria per `part1` →
- Assunzioni:
  - `part1` inizia alla locazione 2000.
  - Interi occupano 4 byte
  - `NAME_LEN` vale 25
  - Non ci sono gap (buchi) fra i vari membri



Dichiarazione di strutture

- Rappresentazione astratta:



- I valori dei membri verranno scritti successivamente nella memoria rappresentata dai riquadri

Dichiarazione di strutture

- Ogni struttura rappresenta un nuovo spazio di nomi (scope)
- I nomi dichiarati all'interno della struttura **non** andranno in conflitto con altri nomi (uguali) dichiarati nel programma
- Nella terminologia del C si dice che ogni struttura ha uno ***spazio dei nomi*** per i suoi membri

Dichiarazione di strutture

- Ad esempio queste dichiarazioni non creano conflitto di nomi:

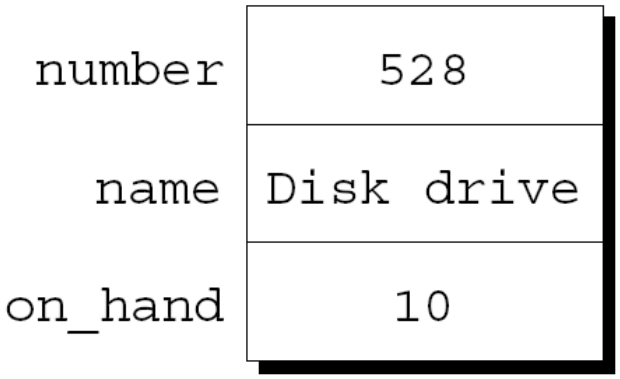
```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;

struct {
  char name[NAME_LEN+1];
  int number;
  char sex;
} employee1, employee2;

int number;
```

Inizializzazione di strutture

- La dichiarazione di strutture può includere un'inizializzazione:

```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1 = {528, "Disk drive", 10},
  part2 = {914, "Printer cable", 5};
```

- `part1` dopo l'inizializzazione:



Inizializzazione di strutture

- Le regole per l'inizializzazione delle strutture sono simili a quelle per gli array
- Le espressioni devono essere costanti
- Un inizializzatore può avere meno membri della struttura che inizializza
  - In tal caso i membri che non vengono inizializzati riceverrano il valore di default 0

Inizializzatori designati (C99)

- Nel C99 si possono usare gli inizializzatori designati
- Rivediamo l'inizializzatore per `part1` dell'esempio precedente:

```
{528, "Disk drive", 10}
```

- In un inizializzatore designato ogni valore viene etichettato non il nome del membro che inizializza:

```
{.number = 528, .name = "Disk drive", .on_hand = 10}
```

- La combinazione del punto e del nome viene detta ***designatore***

Inizializzatori designati (C99)

- Gli inizializzatori designati sono più facili da leggere e da correggere
- Inoltre, usando i designatori, l'ordine non deve necessariamente essere quello in cui i membri sono stati dichiarati nella struttura
  - Il programmatore non deve ricordare l'ordine
  - L'ordine nella dichiarazione può essere cambiato senza che questo comporti modifiche nelle inizializzazioni con i designatori

Operazioni sulle strutture

- Per accedere ad un membro di una struttura scriviamo il nome della struttura prima, poi un punto e poi il nome del membro
- Ecco delle istruzioni che stampano i valori dei membri della struttura `part1`:

```
printf("Part number: %d\n", part1.number);
printf("Part name: %s\n", part1.name);
printf("Quantity on hand: %d\n", part1.on_hand);
```

Operazioni sulle strutture

- I membri di una struttura sono lvalues
- Possono apparire nella parte sinistra di un assegnamento oppure come operando di un incremento o un decremento:

```
part1.number = 258;
  /* changes part1's part number */
part1.on_hand++;
  /* increments part1's quantity on hand */
```

Operazioni sulle strutture

- Il punto usato per specificare un membro della struttura è in effetti un operatore.
- Ha precedenza su praticamente tutti gli altri operatori
- Esempio:

```
scanf("%d", &part1.on_hand);
```

L'operatore `.` ha precedenza sull'operatore `&`, quindi `&` calcola l'indirizzo di `part1.on_hand`.

Operazioni sulle strutture

- L'altra operazione principale sulle strutture è l'assegnamento:

```
part2 = part1;
```

- L'effetto di questa istruzione è la ***copia*** di `part1.number` in `part2.number`, di `part1.name` in `part2.name`, e così via

Operazioni sulle strutture

- Gli array non possono essere copiati con l'operatore di assegnamento =, ma un array all'interno di una struttura viene copiato quando la struttura che lo contiene viene copiata
- Alcuni programmatori sfruttano questa caratteristica creando delle strutture "fittizie" che contengono un array per poterlo copiare con l'assegnamento:

```
struct { int a[10]; } a1, a2;
a1 = a2;
  /* legal, since a1 and a2 are structures */
```

Operazioni sulle strutture

- L'operatore = può essere usato solo con strutture con tipi ***compatibili***
- Strutture dichiarate con la stessa istruzione (come `part1` e `part2`) sono compatibili
- Strutture dichiarate usando la stessa definizione di struttura (vedremo fra poco) sono compatibili
- L'assegnazione è l'unica operazione permessa
- In particolare, non si possono usare gli operatori == e != per confrontare due strutture

Tipo struttura

- Se c'è la necessità di dichiarare molte strutture con lo stesso tipo si può definire un *tipo struttura*
- Il tipo della struttura conterrà solo la definizione della struttura ma non la dichiarazione di una variabile
- Ci sono due possibilità
  - Dichiarare un "nome di struttura"
  - Usare `typedef` per definire un nuovo tipo

Dichiarare un nome di struttura

- Un ***nome di struttura*** è un nome usato per identificare un particolare tipo di struttura
- Nella dichiarazione di una struttura il nome (`part` nell'esempio) precede la definizione della struttura

```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
};
```

- Si noti il punto e virgola alla fine della dichiarazione

Dichiarare un nome di struttura

- Il nome `part` può essere usato per dichiarare nuove variabili strutturate:

```
struct part part1, part2;
```

- Si noti che non si può omettere la parola chiave `struct`:

```
part part1, part2;   /*** WRONG ***/
```

`part` non è il nome di un tipo e senza la parola chiave `struct`, non ha significato

- Poiché i nomi di strutture non vengono riconosciuti se non preceduti da `struct`, essi non vanno in conflitto con altri nomi usati nel programma.

Dichiarare un nome di struttura

- La dichiarazione di un *nome* per la struttura può essere fatta insieme alla dichiarazione di *variabili*

```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;
```

Dichiarare un nome di struttura

- Tutte le strutture dichiarate usando `struct part` sono compatibili fra di loro:

```
struct part part1 = {528, "Disk drive", 10};
struct part part2;

part2 = part1;
  /* legal; both parts have the same type */
```

Definizione di un tipo struttura

- Alternativamente è possibile dichiarare un nuovo tipo usando `typedef`
- Ecco la definizione di un nuovo tipo `Part`:

```
typedef struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} Part;
```

- `Part` adesso può essere usato come gli altri tipi:

```
Part part1, part2;
```

Definizione di un tipo struttura

- Per avere un tipo struttura possiamo sia usare il nome di struttura sia la definizione di un nuovo tipo con `typedef`.

Strutture come argomenti e valori di ritorno

- Le funzioni possono avere strutture come argomenti e come valori di ritorno
- Una funzione con una struttura come argomento:

```
void print_part(struct part p)
{
  printf("Part number: %d\n", p.number);
  printf("Part name: %s\n", p.name);
  printf("Quantity on hand: %d\n", p.on_hand);
}
```

- Una chiamata a `print_part`:

```
print_part(part1);
```

Strutture come argomenti e valori di ritorno

- Una funzione che restituisce una struttura `part`

```
struct part build_part(int number,
                       const char *name,
                       int on_hand)
{
  struct part p;

  p.number = number;
  strcpy(p.name, name);
  p.on_hand = on_hand;
  return p;
}
```

- Una chiamata a `build_part`:

```
part1 = build_part(528, "Disk drive", 10);
```

Strutture come argomenti e valori di ritorno

- Sia passare una struttura ad una funzione che restituire una struttura come valore di ritorno **richiedono la copia** di tutti i membri della struttura
- Per evitare troppe operazioni di copia è preferibile passare un puntatore alla struttura

Array e strutture annidati

- Strutture ed array possono essere combinati senza restrizioni
- Array possono avere strutture come elementi
- Strutture possono contenere array come membri

Strutture annidate

- A volte è utile annidare una struttura in un'altra struttura
- Supponiamo che `person_name` sia la seguente struttura:

```
struct person_name {
  char first[FIRST_NAME_LEN+1];
  char middle_initial;
  char last[LAST_NAME_LEN+1];
};
```

Strutture annidate

- Possiamo usare `person_name` come parte di una struttura più grande:

```
struct student {
  struct person_name name;
  int id, age;
  char sex;
} student1, student2;
```

- Per accedere al nome, all'iniziale del secondo nome o al cognome di `student1`, occorrono due applicazioni dell'operatore punto (.)
- Esempio:

```
strcpy(student1.name.first, "Fred");
```

Strutture annidate

- Usare una struttura `name` per il nome permette di trattare in modo semplice i nomi
- Una funzione che stampa il nome ha bisogno di un solo argomento non 3:

```
display_name(student1.name);
```

- Copiare i nomi richiede un solo assegnamento non 3:

```
struct person_name new_name;
…
student1.name = new_name;
```

Array di strutture

- Spesso si utilizzano array di strutture
- In questo modo si riesce a memorizzare un database (semplice)
- Ecco un array di strutture `part` che può memorizzare informazioni relative a 100 oggetti:

```
struct part inventory[100];
```

Array di strutture

- Per accedere ad un singolo oggetto si può usare l'indice:

```
print_part(inventory[i]);
```

- Per accedere ad un membro della struttura che memorizza informazioni si usa una combinazione di indice e operatore di selezione:

```
inventory[i].number = 883;
```

- Accedere ad un singolo carattere del nome dell'oggetto, richiede l'uso di un primo indice, della selezione del membro e di un secondo indice:

```
inventory[i].name[0] = '\0';
```

Inizializzazione di un array di strutture

- L'inizializzazione di un array di strutture è simile all'inizializzazione di array multidimensionali
- Ogni struttura ha il suo inizializzatore fra parentesi graffe
- L'inizializzatore dell'array racchiude fra graffe gli inizializzatori delle strutture

Inizializzazione di un array di strutture

- Un motivo per inizializzare un array di strutture può derivare dal fatto che le informazioni nelle strutture non cambiano
- Esempio: un array che contiene i prefissi telefonici internazionali
- Gli elementi dell'array sono strutture che contengono un nome (della nazione) ed un prefisso:

```
struct dialing_code {
  char *country;
  int code;
};
```

Inizializzazione di un array di strutture

```
const struct dialing_code country_codes[] =
  {{"Argentina",              54}, {"Bangladesh",       880},
   {"Brazil",                 55}, {"Burma (Myanmar)",   95},
   {"China",                  86}, {"Colombia",          57},
   {"Congo, Dem. Rep. of",   243}, {"Egypt",             20},
   {"Ethiopia",              251}, {"France",            33},
   {"Germany",                49}, {"India",             91},
   {"Indonesia",              62}, {"Iran",              98},
   {"Italy",                  39}, {"Japan",             81},
   {"Mexico",                 52}, {"Nigeria",          234},
   {"Pakistan",               92}, {"Philippines",       63},
   {"Poland",                 48}, {"Russia",             7},
   {"South Africa",           27}, {"South Korea",       82},
   {"Spain",                  34}, {"Sudan",            249},
   {"Thailand",               66}, {"Turkey",            90},
   {"Ukraine",               380}, {"United Kingdom",    44},
   {"United States",           1}, {"Vietnam",           84}};
```

- Le parentesi per ogni singola struttura possono essere omesse

Programma: database di oggetti

- Il programma `inventory.c` illustra l'utilizzo di array e strutture annidati
- Il programma gestisce gli oggetti di un magazzino
- Informazioni sui singoli oggetti vengono memorizzati in un array di strutture
- Il contenuto di ogni struttura è:
  - Codice dell'oggetto
  - Nome
  - Quantità

Programma: database di oggetti

- Le operazioni permesse dal programma sono
  - `i` (insert): inserisce il codice di un oggetto, il nome e la quantità iniziale disponibile
  - `s` (search): dato un codice, stampa le informazioni (nome e disponibilità) dell'oggetto
  - `u` (update): dato un codice, aggiorna la quantità disponibile
  - `p` (print): stampa una tabella con tutte le informazioni nel database
  - `q` (quit): termina l'esecuzione

Programma: database di oggetti

- Un esempio di utilizzo del programma:

```
Enter operation code: i
Enter part number: 528
Enter part name: Disk drive
Enter quantity on hand: 10

Enter operation code: s
Enter part number: 528
Part name: Disk drive
Quantity on hand: 10
```

Programma: database di oggetti

```
Enter operation code: s
Enter part number: 914
Part not found.

Enter operation code: i
Enter part number: 914
Enter part name: Printer cable
Enter quantity on hand: 5

Enter operation code: u
Enter part number: 528
Enter change in quantity on hand: -2
```

Programma: database di oggetti

```
Enter operation code: s
Enter part number: 528
Part name: Disk drive
Quantity on hand: 8

Enter operation code: p
Part Number   Part Name                Quantity on Hand
    528       Disk drive                      8
    914       Printer cable                   5

Enter operation code: q
```

Programma: database di oggetti

- Il programma memorizza le informazioni per ogni tipo di oggetto in una struttura
- Tutte le strutture vengono inserite in un array `inventory`
- La variabile `num_parts` mantiene il numero di oggetti presenti nel database

Programma: database di oggetti

- Ecco la struttura principale del programma:

```
for (;;) {
   prompt user to enter operation code;
   read code;
   switch (code) {
      case 'i': perform insert operation; break;
      case 's': perform search operation; break;
      case 'u': perform update operation; break;
      case 'p': perform print operation; break;
      case 'q': terminate program;
      default:  print error message;
   }
}
```

Programma: database di oggetti

- Sono previste delle funzioni per l'inserimento, la ricerca, l'aggiornamento e la stampa
- Poichè tutte queste funzioni dovranno accedere all'array `inventory` ed a `num_parts`, queste variabili saranno globali
- Il programma verrà diviso in 3 file:
  - `inventory.c` (la parte principale del programma)
  - `readline.h` (contiene il prototipo della funzione `read_line`)
  - `readline.c` (contiene la definizione della funzione `read_line`)

inventory.c

```
/* Maintains a parts database (array version) */

#include <stdio.h>
#include "readline.h"

#define NAME_LEN 25
#define MAX_PARTS 100

struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} inventory[MAX_PARTS];

int num_parts = 0;   /* number of parts currently stored */

int find_part(int number);
void insert(void);
void search(void);
void update(void);
void print(void);
```

```
/**********************************************************
 * main: Prompts the user to enter an operation code,     *
 *       then calls a function to perform the requested   *
 *       action. Repeats until the user enters the        *
 *       command 'q'. Prints an error message if the user *
 *       enters an illegal code.                          *
 **********************************************************/
int main(void)
{
  char code;
  for (;;) {
    printf("Enter operation code: ");
    scanf("%c", &code);
    while (getchar() != '\n')   /* skips to end of line */
      ;
```

```
    switch (code) {
      case 'i': insert();
                break;
      case 's': search();
                break;
      case 'u': update();
                break;
      case 'p': print();
                break;
      case 'q': return 0;
      default:  printf("Illegal code\n");
    }
    printf("\n");
  }
}
```

```
/**********************************************************
 * find_part: Looks up a part number in the inventory     *
 *            array. Returns the array index if the part  *
 *            number is found; otherwise, returns -1.     *
 **********************************************************/
int find_part(int number)
{
  int i;

  for (i = 0; i < num_parts; i++)
    if (inventory[i].number == number)
      return i;
  return -1;
}
```

```
/**********************************************************
 * insert: Prompts the user for information about a new   *
 *         part and then inserts the part into the        *
 *         database. Prints an error message and returns  *
 *         prematurely if the part already exists or the  *
 *         database is full.                              *
 **********************************************************/
void insert(void)
{
  int part_number;

  if (num_parts == MAX_PARTS) {
    printf("Database is full; can't add more parts.\n");
    return;
  }
```

```
  printf("Enter part number: ");
  scanf("%d", &part_number);
  if (find_part(part_number) >= 0) {
    printf("Part already exists.\n");
    return;
  }

  inventory[num_parts].number = part_number;
  printf("Enter part name: ");
  read_line(inventory[num_parts].name, NAME_LEN);
  printf("Enter quantity on hand: ");
  scanf("%d", &inventory[num_parts].on_hand);
  num_parts++;
}
```

```
/**********************************************************
 * search: Prompts the user to enter a part number, then  *
 *         looks up the part in the database. If the part *
 *         exists, prints the name and quantity on hand;  *
 *         if not, prints an error message.               *
 **********************************************************/
void search(void)
{
  int i, number;

  printf("Enter part number: ");
  scanf("%d", &number);
  i = find_part(number);
  if (i >= 0) {
    printf("Part name: %s\n", inventory[i].name);
    printf("Quantity on hand: %d\n", inventory[i].on_hand);
  } else
    printf("Part not found.\n");
}
```

```
/**********************************************************
 * update: Prompts the user to enter a part number.       *
 *         Prints an error message if the part doesn't    *
 *         exist; otherwise, prompts the user to enter    *
 *         change in quantity on hand and updates the     *
 *         database.                                      *
 **********************************************************/
void update(void)
{
  int i, number, change;

  printf("Enter part number: ");
  scanf("%d", &number);
  i = find_part(number);
  if (i >= 0) {
    printf("Enter change in quantity on hand: ");
    scanf("%d", &change);
    inventory[i].on_hand += change;
  } else
    printf("Part not found.\n");
}
```

```
/**************************************************
 * print: Prints a listing of all parts in the database,  *
 *        showing the part number, part name, and         *
 *        quantity on hand. Parts are printed in the      *
 *        order in which they were entered into the       *
 *        database.                                       *
 **************************************************/
void print(void)
{
  int i;

  printf("Part Number   Part Name                    "
         "Quantity on Hand\n");
  for (i = 0; i < num_parts; i++)
    printf("%7d       %-25s%11d\n", inventory[i].number,
           inventory[i].name, inventory[i].on_hand);
}
```

Programma: database di oggetti

- La versione di `read_line` del Capitolo 13 non funzionerà correttamente nel programma che stiamo sviluppando
- Consideriamo cosa accade quando l'utente inserisce un oggetto:

```
Enter part number: 528
Enter part name: Disk drive
```

- L'utente preme "Invio" dopo il codice dell'oggetto, creando un carattere di newline che il programma deve leggere
- Quando si usa la scanf `scanf` verranno "consumati" il 5, il 2, e l'8, ma non il newline

Programma: database di oggetti

- Se proviamo ad usare la `read_line` originale, la funzione si fermerà appena trova il newline
- Questo problema è comune quando un input numerico è seguito da un input di caratteri
- Una soluzione è quella di scrivere una versione di `read_line` che "salta" i caratteri bianchi prima di memorizzare i caratteri non bianchi
- Così si risolve il problema del newline e si evita anche che spazio bianco venga memorizzato come parte del nome dell'oggetto

readline.h

```
/*************************************************
 * read_line: Skips leading white-space characters, then  *
 *            reads the remainder of the input line and   *
 *            stores it in str. Truncates the line if its *
 *            length exceeds n. Returns the number of     *
 *            characters stored.                          *
 *************************************************/
int read_line(char str[], int n);
```

readline.c

```
#include <ctype.h>
#include <stdio.h>
#include "readline.h"

int read_line(char str[], int n)
{
  int ch, i = 0;

  while (isspace(ch = getchar()))
    ;
  while (ch != '\n') {
    if (i < n)
      str[i++] = ch;
    ch = getchar();
  }
  str[i] = '\0';
  return i;
}
```

Unioni

- Una variabile ***unione*** (o semplicemente un'unione), come una struttura, consiste di più membri, non necessariamente dello stesso tipo
- Il compilatore alloca memoria solo per il **membro** dell'unione che richiede lo **spazio maggiore**
- Tutti i membri condividono lo stesso spazio di memoria
- Cambiando il valore di un membro si cambia il valore di tutti gli altri membri (la memoria è la stessa)

Unioni

- Ecco un esempio di un'unione:

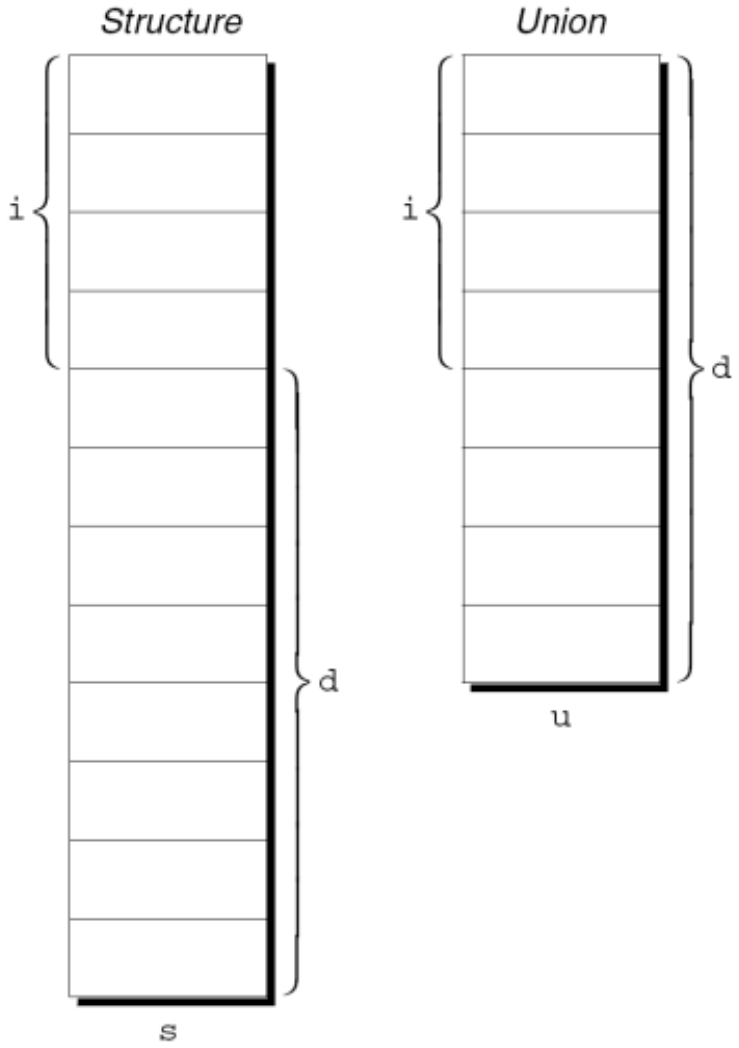
```
union {
   int i;
   double d;
} u;
```

- La dichiarazione è sostanzialmente la stessa di una struttura:

```
struct {
   int i;
   double d;
} s;
```

Unioni

- La differenza fra `s` (struttura) e `u` (unione) è nell'uso della memoria
- I membri di `s` vengono memorizzati in celle di memoria separate
- I membri di `u` vengono memorizzate nelle stesse celle di memoria



Unioni

- Ai membri di un'unione si accede nello stesso modo in cui si accede ai membri di una struttura:

```
u.i = 82;
u.d = 74.8;
```

- Cambiando il valore di un membro si cambia anche il valore degli altri membri
  - Memorizzare un valore `u.d` fa "perdere" il valore precedentemente memorizzato in `u.i`
  - Cambiamenti fatti a `u.i` si ripercuotono in cambiamenti a `u.d`

Unioni

- Le proprietà delle unioni sono praticamente identiche a quelle delle strutture
- Possiamo dichiarare dei nomi di unione o dei tipi di unione come facciamo per le strutture
- Possiamo usare l'operatore di assegnamento = per copiare un'unione
- Possiamo passare un'unione ad una funzione
- Una funzione può restituire un'unione

Unioni

- Solo il primo membro può essere inizializzato
- Ecco un esempio:

```
union {
   int i;
   double d;
} u = {0};
```

- L'espressione fra parentesi graffe deve essere un'espressione costante

Unioni

- Possiamo usare anche i designatori
- Un designatore ci permette di specificare quale membro inzializzare:

```
union {
  int i;
  double d;
} u = {.d = 10.0};
```

- Quindi con il designatore possiamo inizializzare un membro qualsiasi, non per forza il primo. Rimane il vincolo di un solo membro da poter inizializzare

Unioni

- Utilizzo delle unioni
  - Risparmiare memoria
  - Costruire strutture dati composite
  - Utilizzare la stessa zona di memoria in modi diversi (Nel Capitolo 20 viene usata questa possibilità)

Usare le unioni per risparmiare memoria

- Supponiamo di dover utilizzare una struttura che contiene informazioni riguardo un articolo venduto in un catalogo regali
- Gli articoli venduti sono libri, tazze e magliette
- Ogni articolo ha un numero identificativo, un prezzo e altre informazioni che dipendono dall'articolo:

Books (Libri): Titolo, autore, numero di pagine

Mugs (Tazze): Motivo

Shirts (Magliette): Motivo, colori disponibili, taglie disponibili

Usare le unioni per risparmiare memoria

- Un primo tentativo per la struttura `catalog_item`

```
struct catalog_item {
  int stock_number;
  double price;
  int item_type;
  char title[TITLE_LEN+1];
  char author[AUTHOR_LEN+1];
  int num_pages;
  char design[DESIGN_LEN+1];
  int colors;
  int sizes;
};
```

Usare le unioni per risparmiare memoria

- Il valore di `item_type` potrà essere uno fra `BOOK`, `MUG`, o `SHIRT`.
- Le variabili `colors` e `sizes` codificheranno combinazioni di colori e taglie
- Questa struttura spreca molto spazio poichè solo parte delle informazioni è necessaria per tutti gli articoli
- Possiamo ridurre l'uso di memoria usando un'unione all'interno della struttura `catalog_item`

Usare le unioni per risparmiare memoria

```
struct catalog_item {
  int stock_number;
  double price;
  int item_type;
  union {
    struct {
      char title[TITLE_LEN+1];
      char author[AUTHOR_LEN+1];
      int num_pages;
    } book;
    struct {
      char design[DESIGN_LEN+1];
    } mug;
    struct {
      char design[DESIGN_LEN+1];
      int colors;
      int sizes;
    } shirt;
  } item;
};
```

Usare le unioni per risparmiare memoria

- Se `c` è una struttura `catalog_item` che rappresenta un libro, allora potremo accedere al titolo del libro nel seguente modo:

```
printf("%s", c.item.book.title);
```

Usare le unioni per strutture dati miste

- Le unioni possono essere usate per creare strutture dati che contengono tipi assortiti
- Supponiamo di voler creare un array di elementi che siano `int` oppure `double`
- Come prima cosa definiamo un'unione che permette di accedere alla memoria sia come `int` che come `double`:

```
typedef union {
  int i;
  double d;
} Number;
```

Usare le unioni per strutture dati miste

- Quindi creiamo un array di variabili di tipo `Number`:

```
Number number_array[1000];
```

- Il tipo `Number` può memorizzare sia un `int` che un `double`
- Questo permette di avere `int` e `double` nell'array `number_array`:

```
number_array[0].i = 5;
number_array[1].d = 8.395;
```

Usare un'etichetta

- Il C non prevede nessun modo per determinare con quale membro il valore di un'unione è stato cambiato
  - Se usiamo il valore con un membro accedendo con un altro membro il valore potrebbe non avere significato
- Consideriamo di nuovo l'unione `Number` e supponiamo di volere scrivere una funzione che stampi il valore contenuto nell'unione:

```
void print_number(Number n)
{
  if (n contains an integer)
    printf("%d", n.i);
  else
    printf("%g", n.d);
}
```

- Non c'è modo di sapere se `n` contiene un `int` o un `double`

Usare un'etichetta

- Per ovviare al problema possiamo mantenere questa informazione in modo esplicito aggiungendo un membro all'unione per memorizzare l'informazione che ci interessa
- `item_type` ha svolto questa funzione per la struttura `catalog_item`

Usare un'etichetta

- Ecco un esempio per `Number` che diventa una struttura con all'interno un'unione:

```
#define INT_KIND 0
#define DOUBLE_KIND 1

typedef struct {
  int kind;   /* tag field */
  union {
    int i;
    double d;
  } u;
} Number;
```

- Il valore del membro `kind` sarà `INT_KIND` oppure `DOUBLE_KIND`.

Usare un'etichetta

- Ogni volta che cambieremo il valore dell'unione `u`, assegneremo l'appropriato valore a `kind` per ricordarci quale membro di `u` è stato modificato
- Ecco un esempio:

```
n.kind = INT_KIND;
n.u.i = 82;
```

dove `n` è una variabile di tipo `Number`

Usare un'etichetta

- Quando accediamo al valore memorizzato nella variabile di tipo `Number`, la variabile `kind` ci dirà quale membro dell'unione usare per recuperare il valore corretto
- Ecco la funzione `print_number` che sfrutta l'etichetta che abbiamo aggiunto:

```
void print_number(Number n)
{
  if (n.kind == INT_KIND)
    printf("%d", n.u.i);
  else
    printf("%g", n.u.d);
}
```

Enumerazioni

- In alcuni casi c'è bisogno di usare variabili che assumono solo valori presi da un insieme molto piccolo
- Ad esempio una variabile che memorizza il seme di una carta da gioco può assumere solo 4 valori: "cuori", "quadri", "fiori" e "picche"

Enumerazioni

- Ovviamente si può usare un `int` per una variabile di tipo "seme" ed associare un particolare valore ad ogni possibile seme:

```
int s;    /* s will store a suit */
…
s = 2;    /* 2 represents "hearts" */
```

- Svantaggi:
  - Non è immediato capire che `s` può assumere solo 4 valori
  - Il significato (es. cuori) di uno specifico valore (es. 2) non è evidente

Enumerazioni

- Potremmo usare delle macro per migliorare la situazione:

```
#define SUIT      int
#define CLUBS     0
#define DIAMONDS  1
#define HEARTS    2
#define SPADES    3
```

- Il frammento del nostro programma diventerebbe:

```
SUIT s;
…
s = HEARTS;
```

Enumerazioni

- È un miglioramento ma:
  - Non c'è indicazione per chi guarda il codice che le 4 macro rappresentano valori dello stesso "tipo"
  - Se i valori possibili non sono proprio pochi, definire una macro per ogni possibile valore potrebbe diventare tedioso
  - I nomi `CLUBS`, `DIAMONDS`, `HEARTS`, e `SPADES` saranno rimossi dal preprocessore, quindi non saranno più disponibili per il debug

Enumerazioni

- Il C fornisce un tipo speciale progettato appositamente per queste necessità
- Un ***tipo enumerato (enumerazione)*** è un tipo per il quale i possibili valori vengono elencati ("enumerati") dal programmatore
- Ogni valore deve avere un nome (una ***costante di enumerazione***).

Enumerazioni

- Sebbene le enumerazioni abbiano poco in comune con strutture ed unioni, sono dichiarate in modo simile:

```
enum {CLUBS, DIAMONDS, HEARTS, SPADES} s1, s2;
```

- I nomi usati nell'enumerazione devono essere diversi da tutti gli altri identificatori
  - Non devono andare in conflitto con altri nomi di variabili

Enumerazioni

- Le enumerazioni costanti sono molto simili ai valori creati con le macro ma non sono equivalenti
- Se un'enumerazione è dichiarata all'interno di una funzione, le sue costanti **non** saranno visibili al di fuori della funzione.

Enumerazioni

- Come per le strutture e le unioni, ci sono due modi per dare un "nome" alle enumerazioni: dichiarando un'etichetta oppure definendo un tipo con `typedef`
- Le etichette sono simili a quelle delle unioni e delle strutture:

```
enum suit {CLUBS, DIAMONDS, HEARTS, SPADES};
```

- variabili `suit` possono essere dichiarate in questo modo:

```
enum suit s1, s2;
```

Enumerazioni

- In alternativa si può definire un nuovo tipo:

```
typedef enum {CLUBS, DIAMONDS, HEARTS, SPADES} Suit;
Suit s1, s2;
```

- Nel C89, usare `typedef` per creare un'enumerazione per il tipo Booleano è un'ottima idea:

```
typedef enum {FALSE, TRUE} Bool;
```

Enumerazioni ed interi

- In realtà il C tratte le costante delle enumerazioni come interi
- Normalmente il compilatore assegna i valori ai nomi dell'enumerazioni partendo da 0, 1, 2, … e così via
- Nell'esempio dell'enumerazione `suit` i nomi `CLUBS`, `DIAMONDS`, `HEARTS`, e `SPADES` corrispondono, ripsettivamente, ai valori 0, 1, 2, e 3.

Enumerazioni ed interi

- È possibile specificare in modo esplicito i valori che si vogliono usare:

```
enum suit {CLUBS = 1, DIAMONDS = 2,
           HEARTS = 3, SPADES = 4};
```

- I valori possono essere interi qualsiasi, in un qualsiasi ordine:

```
enum dept {RESEARCH = 20,
           PRODUCTION = 10, SALES = 25};
```

- È perfino possibile usare lo stesso valore per due nomi

Enumerazioni ed interi

- Quando non viene specificato, il valore assegnato è pari al valore precedentemente utilizzato più 1
- Se non è specificato il valore della prima costante, il compilatore usa il valore 0
- Esempio:

```
enum EGA_colors {BLACK, LT_GRAY = 7,
                 DK_GRAY, WHITE = 15};
```

`BLACK` vale 0, `LT_GRAY` vale 7, `DK_GRAY` vale 8, e `WHITE` vale 15.

Enumerazioni ed interi

- Poichè di fatto sono interi, i nomi delle enumerazioni possono essere usati come interi:

```
int i;
enum {CLUBS, DIAMONDS, HEARTS, SPADES} s;

i = DIAMONDS;    /* i is now 1                */
s = 0;           /* s is now 0 (CLUBS)        */
s++;             /* s is now 1 (DIAMONDS)     */
i = s + 2;       /* i is now 3                */
```

- `s` viene trattata come una variabile intera.
- `CLUBS`, `DIAMONDS`, `HEARTS`, e `SPADES` sono nomi alternativi per gli interi 0, 1, 2, e 3

Enumerazioni ed interi

- Sebbene poter usare le costanti di un'enumerazione come interi può a volte essere conveniente, è pericoloso memorizzare un valore intero (esplicito) in una variabile che è un'enumerazione
  - Il valore potrebbe non corrispondere a nessuno degli interi elencati nell'enumerazione
- Ad esempio, se memorizzassimo accidentalmente il valore 4 nella variabile `s`, successivamente potremmo avere problemi con quella variabile

Enumerazioni e campi etichetta

- Le enumerazioni sono perfette per determinare qual è il campo dell'unione che è stato utilizzato per assegnare il valore all'unione
- Riprendendo l'esempio della struttura `Number` possiamo utilizzare un'enumerazione per `kind`

```
typedef struct {
  enum {INT_KIND, DOUBLE_KIND} kind;
  union {
    int i;
    double d;
  } u;
} Number;
```

Enumerazioni e campi etichetta

- La nuova struttura è equivalente alla precedente
- Vantaggi:
  - Non ci servono più le macro per `INT_KIND` e `DOUBLE_KIND`
  - Rende evidente il fatto che `kind` può assumere solo due valori: `INT_KIND` e `DOUBLE_KIND`

… arrivederci alla prossima lezione