Strutture

Struct

- Aggregati **non ordinati** di elementi **di qualsiasi tipo**, riferibili come entità uniche.
  - Definiti elencando gli elementi (detti “campi”).
  - Corrispondono ai record Pascal o Ada.
- La dichiarazione ha la forma:

  `struct` [*identificatore*] `{` *lista di campi* `}`
  - La lista di campi è formata da dichiarazioni, con la stessa sintassi delle dichiarazioni di variabili, ma senza classe di memoria e inizializzazione.

Esempio di struct

```
struct  account
  {
  unsigned int      id;
  char*             name;
  char*             address;
  long              balance;
  };
```

Nome delle struct

- Se l'identificatore manca, il tipo resta anonimo e non può essere riutilizzato altrove.
  - Serve nelle definizioni di tipi con `typedef`.
- Il nome del tipo è `struct` <*identificatore*>: la parola chiave `struct` **fa parte del nome**.

Vincoli di allineamento (1)

- Ogni tipo richiede che le sue istanze (variabili, campi) siano a indirizzi macchina multipli di una costante.
  - Di solito una piccola potenza di 2: 1, 2, 4, 8, 16.
- I vincoli sono imposti dall'h/w.
  - Possono dipendere dalle opzioni di compilazione.
- Il tipo `char` ha il **minimo** requisito di allineamento, vale a dire che le variabili `char` possono essere allocate a qualsiasi indirizzo.

Vincoli di allineamento (2)

- Non rispettandoli, quando si accede alla variabile si può avere (a seconda della macchina):
  - l’arresto in errore del programma;
  - un valore imprevedibile;
  - una perdita di prestazioni.

Esempi di vincoli di allineamento

- Su PA-RISC (HP), Alpha e SPARC `int`, `long` e `float` devono essere a indirizzi multipli di 4, i `double` a indirizzi multipli di 8, altrimenti si ha un segnale di indirizzo illegale (SIGADR).
- Su Intel x86 `int`, `long`, `float` e `double` devono essere a indirizzi multipli di 4, altrimenti le prestazioni peggiorano.

Dimensione delle struct (1)

- In memoria i campi di una `struct` sono **ordinati**, ma non necessariamente **consecutivi**.
  - Compilatori differenti possono avere comportamenti differenti **sulla stessa macchina**.
  - Se il codice ipotizza la consecutività dei campi non è portabile.
- La dimensione di una `struct` è **maggiore o uguale** alla somma delle dimensioni dei campi.
  - I compilatori **possono inserire** campi di riempimento per rispettare i **vincoli di allineamento**.

Dimensione delle struct (2)

- Per esempio, la dimensione della `struct` seguente è 12 o 16, su molte macchine, anche se la somma delle dimensioni dei campi è 9:

```
struct
  {
  char      ch;
  double    d;
  };
```

Accesso ai campi di struct

- Si può trattare una `struct` come un'entità unica, trattandola come una qualsiasi variabile.
- Si può accedere a un singolo campo con l'operatore `.` tramite la sintassi:

  <struttura> `.` *<nome di campo>*

  – L'operatore `.` ha la massima priorità ed è associativo da sinistra: `a.b.c` equivale a `(a.b).c`.

Esempio di accesso ai campi di struct

```
struct date_type
  {
  unsigned int    day;
  unsigned int    month;
  int             year;
  } today;

today.day = 12;
if (today.year >= 2019)
```

Operazioni ammesse sulle struct

- Sono permessi **solo** i seguenti utilizzi di `struct`:
  - accesso ai singoli campi;
  - assegnamento a entità dello stesso tipo;
  - passaggio come argomento a funzioni che dichiarino argomenti formali dello stesso tipo;
  - produzione come valori di funzione.
- In particolare dichiarando una `struct` contenente un vettore:
  - lo si può passare a una funzione, **copiando** tutti i valori.
  - si può far produrre a una funzione un vettore.

Struct come argomenti (1)

- Le `struct` possono essere passate come argomenti alle funzioni. P. es.:

```
double  distance(struct point p1,
                 struct point p2)
   {
   return sqrt((p1.x - p2.x) *
        (p1.x - p2.x) +
        (p1.y - p2.y) * (p1.y - p2.y));
   }
```

Struct come argomenti (2)

- Attenzione però alla differenza: passando `struct` per valore, invece che tramite puntatori, queste vengono **copiate** nel frame della funzione.
  - Occupano **più memoria** che passando puntatori.
    - Con strutture grandi e funzioni ricorsive si rischia lo stack overflow.
  - La copia richiede **tempo**.
  - L'accesso è **più veloce**.
    - Si risparmia l'indirettezza del puntatore.

Passaggio per valore di vettori

- Si realizza inserendo vettori o matrici in una `struct` e passandola per valore:
  - Si consuma tempo e memoria.
  - Le dimensioni del vettore o matrice devono essere **fisse**.
  - Es.:

```
struct envelope
   {
   int  a [100];
   }

void    f(struct envelope arg)
   { … }
```

Funzioni che producono struct

- Non pongono particolari problemi.
  - Per produrre il valore si utilizza di solito una variabile locale temporanea o un valore composto.
  - Il valore prodotto viene trasferito nel frame del chiamante uscendo dalla funzione.

Esempio di funzioni che producono struct

```
struct point   { int x, y; };

struct point     makepoint(int x, int y)
  {
  struct point   temp;

  temp.x = x;
  temp.y = y;
  return temp;
  }
…
p1 = makepoint(0, 0);
```

Attenzione al valore prodotto (1)

- Dichiarando:

  ```
  struct point* f(void);
  ```

  si dichiara una funzione che produce come valore un **puntatore** a una `struct`.
  - La funzione deve produrre un puntatore a una variabile statica o dinamica, ma **non automatica** (locale), che viene distrutta all'uscita.
    - La funzione produrrebbe un puntatore non utilizzabile, ma senza segnalazione d'errore.

Attenzione al valore prodotto (2)

Una funzione come la seguente è **corretta**:

```
struct point f(void)
    {
    struct point      temp;
    …
    return temp;      Il valore viene copiato
    }
```

Mentre è un **errore** una funzione come questa:

```
struct point* f(void)
    {
    struct point      temp;
    …
    return &temp;     Il puntatore non è utilizzabile dal chiamante
    }
```

Funzioni che producono vettori

- Come per il passaggio di argomenti, si realizzano inserendo vettori o matrici in una `struct`:
  - Le dimensioni del vettore o matrice devono essere **fisse**.
  - Es.:

```
struct envelope
   {
   int  a [100];
   }

struct envelope   f(void)
   { … }
```

Inizializzazione di struct (1)

- Si inizializzano come i vettori, con una lista di valori separati da virgole e racchiusi tra parentesi graffe.
- Se i valori sono troppi, si ha un errore.
- Se i valori sono pochi, gli elementi mancanti sono inizializzati a zero.
- I valori tra graffe sono convertiti ai tipi dei campi con le regole degli assegnamenti.

Inizializzazione di struct (2)

Per esempio:

```
struct date_type
  {
  unsigned int  day;
  unsigned int  month;
  int           year;
  } today = { 26, 3, 2019 };
```

Assegnamenti di strutture

- Possibili tra `struct` dello stesso tipo con l'operatore =.
  - Per esempio:

```
struct xyz      a, b;
a = b;
```

- Non è permesso il confronto con operatori come `==` e `!=`: bisogna confrontare campo per campo.
  - Per esempio, è illegale:

```
if (a == b)
```

Confronti di puntatori a campi (1)

- E’ ammesso confrontare tra loro puntatori a campi differenti, ma **dello stesso tipo**, di una stessa `struct`.
  - Appare maggiore il puntatore al campo dichiarato dopo.
  - L’operazione **non ha alcuna utilità pratica**.
- Operazioni di incremento o decremento per passare da un campo a un altro non sono portabili.

Confronti di puntatori a campi (2)

- Dichiarando:

```
struct
    {
    int   c1;
    int   c2;
    int   c3;
    } s;

int*      p1 = &s.c1;
int*      p2 = &s.c2;
```

  È sempre vero che `p2 > p1`, ma non è sempre vero che `p1 + 1 == p2`.

Strutture complesse

- Le `struct` possono essere combinate a piacere nelle strutture dati; si possono avere:
  - vettori di `struct`;
  - `struct` contenenti vettori;
  - `struct` contenenti `struct`;
  - …
- L'accesso agli elementi si effettua combinando operatori `.` e `[]`.
- Nelle inizializzazioni si utilizza un livello di parentesi graffe per ogni livello di `struct` e dimensione di vettore.

Esempi di strutture complesse (1)

```
struct point
  {
  int  x;
  int  y;
  };
struct rectangle
  {
  struct point   p1;
  struct point   p2;
  };
struct rectangle v [100];
v [n].p1.x = 0;
```

Esempi di strutture complesse (2)

```
struct entry
  {
  const char*     name;
  unsigned int    count;
  };
struct entry      table [] =
  {
        { "Milano", 0 },
        { "Bari", 0 },
        { "Torino", 0 },
        { "Roma", 0 }
  };
```

Operatore `->`

- Serve ad accedere a un campo di una `struct` puntata da un puntatore.
  - Equivale a una combinazione degli operatori `*` (unario) e `.`.
- Come l'operatore `.` ha la massima priorità ed è associativo da sinistra: `a->b->c` equivale a `(a->b)->c`.

Esempio di operatore `->`

```
struct point
  {
  int  x;
  int  y;
  };
struct point*    p;
```

`p->x` equivale a `(*p).x`.

`++p->x` equivale a `++(p->x)` e incrementa il campo `x`.

`p->x++` equivale a `(p->x)++` e incrementa il campo `x`.

`(++p)->x` incrementa il puntatore e accede al campo `x`.

`p++->x` incrementa il puntatore e accede al campo `x`.

Vettori “flessibili” (1)

- L'ultimo campo di una `struct` può essere un vettore di dimensione non specificata.
  - Assegnamenti tra entità di tali `struct` ignorano il vettore.
  - L'operatore `sizeof` applicato a tali `struct` ignora il vettore.

Vettori “flessibili” (2)

- Non si possono definire variabili statiche o automatiche `struct` contenenti vettori flessibili.
  - Variabili di tali `struct` possono essere istanziate solo con le funzioni di allocazione dinamica della memoria (`malloc` e simili), precisando la dimensione del vettore.
  - Puntatori a variabili di tali `struct` possono essere trattati normalmente e in particolare passati come argomento e prodotti come valori di funzioni.

Esempio di struct con vettore flessibile

```
struct video_message
  {
  unsigned int      x;
  unsigned int      y;
  unsigned int      length;
  char              message [];
  } *p;

p = (struct video_message *)
  malloc(sizeof(struct video_message) +
  n * sizeof(char));
```

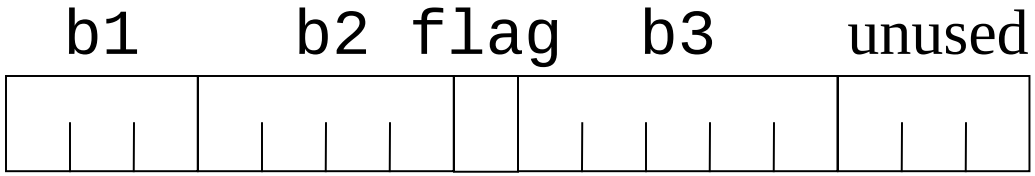
- Alloca una variabile di tipo `video_message`, con vettore `message` di lunghezza `n`.

Bit field

- Campi di `struct`, che hanno dimensione specificata in bit.
  - Al nome del campo si fa seguire il carattere `:` e la dimensione, che deve essere una costante intera.
- Devono essere dichiarati di tipo `_Bool`, `int`, `signed int` o `unsigned int`.
  - Dichiararli `int` non è sicuro, perché in tal caso **dipende dall'implementazione** il fatto che siano con segno o meno.

Esempio di bit-field

```
struct s
  {
  unsigned int    b1: 3;    // from 0 to 7
  signed int      b2: 4;    // from –7 to 7
  _Bool           flag: 1   // true or false
  int             b3: 5;    // ?
  };
```



Attenzione! Non è detto che `b3` possa rappresentare i valori -1 o 15: può essere `signed` o `unsigned`.

Restrizioni sui bit field (1)

- Ai bit field non si può applicare l'operatore (unario) &.
  - Vi si accede **esclusivamente** con gli operatori `.` e `->`.
- Non si possono dichiarare vettori di bit field o puntatori a bit field.
- La loro dimensione massima è **almeno** la dimensione della parola della macchina.
  - In pratica, `CHAR_BIT * sizeof(int)`.
  - Comunque il limite massimo di lunghezza è almeno 16.

Allocazione dei bit field (1)

- Qualunque ipotesi sull'effettiva allocazione rende il codice **non portabile**.
  - L’allocazione può iniziare dal bit più significativo di una parola o dal meno significativo.
- Non è possibile imporre che attraversino confini di "parola" della macchina; il compilatore può permetterlo o meno.
  - La dimensione della parola dipende dall’implementazione.
- È possibile imporre che **non attraversino** confini di parola, dichiarando un campo senza nome di lunghezza zero.

Allocazione dei bit field (2)

- Non è possibile imporre che siano contigui.
  - Il compilatore può inserire riempitivi anonimi, non utilizzati.
- È possibile imporre che **non siano contigui**, dichiarando campi "riempitivi" senza nome.
  - I riempitivi si utilizzano solamente per imporre una corrispondenza dei campi con registri hardware; in tal caso non ci si preoccupa della portabilità.

Esempio di bit-field con vincoli di allineamento

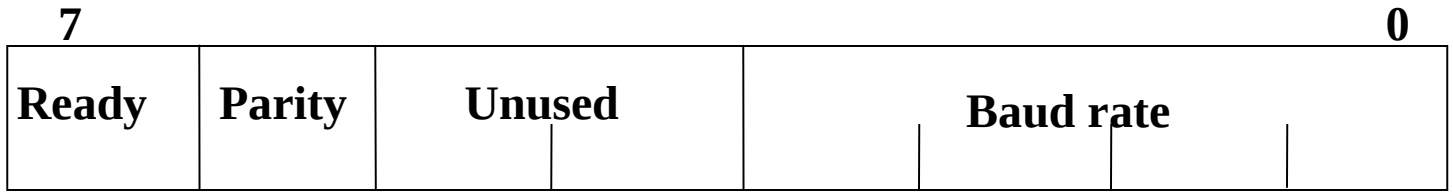
```
struct s
  {
  unsigned int    bl: 5;
                  : 3; // riempitivo anonimo
  signed int      b2: 5;
                  : 0;
        // forzo allineamento alla
        // parola successiva
  unsigned int    b3: 4;
  };
```

Uso corretto dei bit field

- Servono principalmente per risparmiare memoria.
- L'accesso è generalmente **molto più costoso** rispetto ai normali campi di `struct`.
  - Hanno quindi senso solo in **grandi** strutture di dati.
- Possono servire per descrivere registri hardware, in modo **non portabile**.

Bit field per gestione h/w

Dato un registro di una periferica così strutturato:



Possiamo descriverlo così:

```
struct
  {
  _Bool          ready: 1;
  unsigned int   parity: 1;
                 : 2;
  unsigned int baud_rate: 4;
  } *serial_interface;
```

Il fatto che l'allocazione inizi dal bit più significativo o da quello meno significativo, come pure l'inserimento di eventuali riempitivi, dipende dal compilatore.

Unioni

Union (1)

- Unioni **di tipi qualsiasi**.
  - Servono per definire variabili che possono assumere (in istanti diversi) valori di tipo differente.
  - Equivalenti ai record con varianti del Pascal.
- I campi di una `union` sono **in alternativa**, sovrapposti tra loro, e iniziano allo stesso indirizzo.
  - Attenzione: se si assegna un valore a una variante, tutte le altre diventano **indefinite**.

Union (2)

- Si dichiarano come le `struct`, ma con la parola chiave `union` al posto di `struct`.
- Esempio:

```
union u
   {
   int    ival;
   double dval;
   char*  sval;
   };
```

Union (3)

- Per l’accesso ai campi si utilizzano gli operatori `.` e `->`.
- Esempio:

```
union u      a;
union u*     p;

a.ival = 3;
p = &a;
p->sval = "ciao";
```

equivale a:

```
(*p).dval = "ciao";
```

L’ultimo assegnamento rende `a.ival` e `a.dval` indefiniti, ma non ci sono controlli che vietino l’accesso.

Esempio di union

```
struct symbol_table_entry
  {
  enum { INT, DOUBLE, STRING } type;
  union
        {
        int         int_value;
        double      double_value;
        char*       string_value;
        } value;
  };
```

- Viene allocato un unico campo valore, che può contenere valori di tipo differente in istanze diverse.

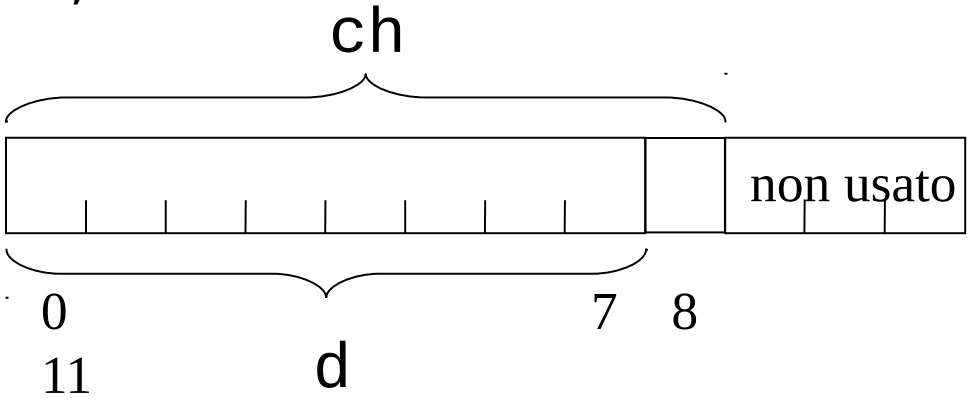
Dimensione delle union (1)

- La dimensione di una `union` è **maggiore o uguale** alla massima dimensioni dei campi (il compilatore può generare un campo di riempimento alla fine, per rispettare i vincoli di allineamento).

Dimensione delle union (2)

- Per esempio, su molte macchine, la dimensione della `union` seguente è 12 o 16, anche se il massimo delle dimensioni dei campi è 9:

```
union
    {
    char        ch [9];
    double      d;
    };
```



Operazioni ammesse sulle union

- Sulle union sono permesse le stesse operazioni permesse sulle `struct`:
  - accesso alle singole varianti;
  - assegnamento a entità dello stesso tipo;
  - passaggio come argomento a funzioni che dichiarino argomenti formali dello stesso tipo;
  - produzione come valori di funzione.

Uso corretto delle union

- Servono a risparmiare memoria, in **grandi** strutture dati.
- Possono servire per passare argomenti di tipi diversi a una funzione.
- Possono servire per far produrre valori di tipi diversi a una funzione.

Esempio di funzione con argomenti di tipo diverso (1)

```
union    number
  {
  int    i;
  long   l;
  double d;
  };

enum kind { INT, LONG, DOUBLE };
```

Esempio di funzione con argomenti di tipo diverso (2)

```
void print(union number n, enum kind k)
  {
  switch (k)
           {
           case INT:
                 (void)printf("%d\n", n.i);
                 return;
           case LONG:
                 (void)printf("%ld\n", n.l);
                 return;
           case DOUBLE:
                 (void)printf("%g\n",n.d);
                 return;
           }
  }
```

Uso scorretto delle union (1)

```
union
  {
  int     i;
  char    c [sizeof(int)];
  } u;

u.i = expression;
x = u.c [sizeof(int) - 1];
```

Uso scorretto delle union (2)

- È un tentativo di accesso al byte meno significativo, ma dipende dall'implementazione ed è **assolutamente da evitare**.

- Il codice corretto:

```
x = espression % (UCHAR_MAX + 1);
```

  è oltretutto generalmente più efficiente.

Inizializzazione delle union

- Si può inizializzare una `union`, con la sintassi usata per `struct`.
  - Se il campo non è specificato, viene inizializzata **la prima variante**: il tipo del valore usato per inizializzare non conta.

```
union
   {
   int       i;
   double    d;
   } u1 = { 10.5 }, u2 = { .d = 3.5 };
```

`u1.i` vale 10, `u2.d` vale 3.5; `u1.d` e `u2.d` sono indefiniti.

Valori compositi

Valori composti (1)

- Formati da un nome di tipo tra parentesi, seguito da inizializzatori tra graffe.
  - Se non usati per inizializzare variabili statiche, gli inizializzatori possono non essere costanti.
  - Eventuali campi non inizializzati hanno valore zero.

Valori compositi (2)

- Sono utilizzabili ovunque sia legale un oggetto del tipo corrispondente:
  - in assegnamenti;
  - in un'istruzione `return`;
  - come argomenti;
  - per inizializzare variabili.

Esempio di valori composti (1)

```
typedef struct
  {
  float x;
  float y;
  } point;

point middle(point p1, point p2)
  {
  return (point){ (p1.x + p2.x) / 2,
        (p1.y + p2.y) / 2 };
  }
```

Esempio di valori composti (2)

```
point   p;

q = middle(p, (point){ 0.0, 0.0 });

p = (point){ x, y };

int*pi = (int []){ 2, 4 };
```

Espressioni nei valori composti

- Nell'inizializzazione di variabili **statiche**, le espressioni che compaiono nei valori composti devono essere **costanti**.
- In tutti gli altri casi possono essere espressioni qualsiasi, anche contenenti chiamate di funzione.
  - Incluso il caso di inizializzazione di variabili locali.

Specifica dei campi nei valori composti

- Nei valori composti si può specificare quale elemento si desidera inizializzare:
  - per i vettori, specificando un indice **costante** tra parentesi quadre;
  - per `struct` e `union`, scrivendo un punto seguito dal nome del campo.
- Alla specifica del campo si fa seguire `=` e il valore che si vuole assegnare.
- In assenza di specifiche, sono inizializzati i campi e gli elementi successivi all'ultimo specificato.

Esempio di specifica dei campi (1)

```
typedef struct
  {
  float      x;
  float      y;
  } point;

point  middle(point *p1, point *p2);

q = middle(p,
        (point){ .x = 0.0, .y = 0.0 });
```

Esempio di specifica dei campi (2)

Le seguenti inizializzazioni di vettori sono equivalenti:

```
int  a [8] = { 0, 1, 2, 3, 0, 0, 0, 9 };
int  a [8] = { [7] = 9, [1] = 1, 2, 3 };
int  a [8] = { [1] = 1, 2, 3, [7] = 9 };
```

Le seguenti inizializzazioni di `struct` sono equivalenti:

```
struct date d = { 1, 2, 2000 };
struct date d = { .day = 1, 2, 2000 };
struct date d =
  { .day = 1, .month = 2, .year = 2000 };
struct date d =
  { .year = 2000, .month = 2, .day = 1 };
```

Pericoli dei valori composti (1)

- Se un campo è inizializzato più volte, **prende l'ultimo valore**, pertanto errori di questo genere non sono rilevati:

```
int a [10] = { [1] = 1, 2, [0] = 3, 4 };
```

`a [1]` vale 4.

```
struct date_type   d =
   { .year = 2000, .month = 12, 2001 };
```

`d.year` vale 2001.

Pericoli dei valori composti (2)

- L'ordine di valutazione delle espressioni è **indefinito**.
  - P. es., se `i` vale 0:

```
int      a [10] = { i, i++ };
```

`a [0]` può essere 0 o 1; `a [1]` vale 0.

  - P. es., se `i` vale 0:

```
struct date_type   start = { i, i, i++ };
```

`start.day` può essere 0 o 1; `start.year` vale 0.

Allocazione dei valori composti

- Un valore composto viene allocato come variabile locale senza nome.
  - Se l’istruzione che lo contiene viene rieseguita, non viene allocata una nuova variabile.

Particolarità dei valori composti

- Dopo l'ultimo valore, prima della graffa chiusa, è ammessa una virgola:

```
long  perfect [] = { 6, 28, 496, };
```

- Si sconsiglia però l'utilizzo di tale possibilità, che è stata introdotta per semplificare il lavoro a generatori automatici di codice e tabelle.