Introduzione alla programmazione in C

Appunti delle lezioni di Complementi di Programmazione

Giorgio Grisetti
Luca Iocchi
Daniele Nardi
Fabio Patrizi
Alberto Pretto

Dipartimento di Ingegneria Informatica, Automatica e Gestionale
Facoltà di Ingegneria dell'Informazione, Informatica, Statistica
Università di Roma "La Sapienza"

Edizione 2022/2023

2023

Indice

7. Tipi di dato strutturati 197

7.1. Record 197

7.1.1. Definire variabili di tipo record con struct 197

7.1.2. Creare nuovi tipi di dato record con struct 199

7.1.3. Tag di struttura 200

7.1.4. Definire tipi di dato record con typedef 201

7.1.5. Variabili di tipo record 203

7.1.6. Accesso ai campi di un record 204

7.1.7. Inizializzazione di variabili di tipo record 204

7.1.8. Assegnazione e uguaglianza 205

7.1.9. Puntatori a record 208

7.1.10. Allocazione dinamica e deallocazione di record 212

7.1.11. Record per la rappresentazione di matrici 214

7.1.12. Record annidati 214

7.1.13. Record autoreferenziali 215

7.1.14. Array di record 216

7.2. Unioni 217

7.3. Tipi di dato enumerati 219

7. Tipi di dato strutturati

7.1. Record

Supponiamo di dover realizzare un'applicazione per manipolare i dati anagrafici di persone: nome, cognome e data di nascita. Sarebbe molto comodo poter trattare ciascuna persona come un'unica variable contenente due campi stringa per memorizzare nome e cognome, e tre campi di tipo intero (short) per memorizzare giorno, mese e anno di nascita. Il tipo di questa variabile sarebbe concettualmente simile ad un array (di cinque elementi), ma con un'importante differenza: i tipi in essa contenuti non sono omogenei. Strutture dati di questo tipo vengono chiamate *record*.

7.1.1. Definire variabili di tipo record con struct

In C è possibile definire variabili di tipo record tramite il costrutto struct come segue:

Dichiarazione di variabile di tipo record

Sintassi:

```
struct {
    dichiarazione-variabile-1;
    ...
    dichiarazione-variabile-n;
} lista-variabili;
```

dove:

- *dichiarazione-variabile-i* è la dichiarazione dell'*i*-esimo campo del record, fornita secondo la sintassi per la dichiarazione di variabili;
- *lista-variabili* è la lista delle variabili di tipo record che si vuole dichiarare.

Semantica:
Vengono create le variabili *menzionate* in *lista-variabili* e, per ciascuna di esse, allocata memoria per un un record avente come membri i campi dichiarati in *dichiarazione-variabile-i*, $i \in 1, \ldots, n$.

Esempio: Il seguente frammento di codice mostra la dichiarazione di due variabili che potrebbero essere usate per memorizzare le informazioni rilevanti di altrettante persone.

```
struct{
    char* nome;
    char* cognome;
    short int giorno;
    short int mese;
    short int anno;
} persona_1, persona_2;
```

Le variabili persona_1 e persona_2 possono essere usate a partire dal momento della loro dichiarazione.

Anche nel caso di variabili di tipo record, come per le variabili di tipo primitivo, la dichiarazione ha per effetto l'allocazione della memoria necessaria a contenere valori del tipo della variabile. Nel caso dei record, la memoria allocata è un'area contigua, la cui dimensione è sufficiente a contenere tutti i campi presenti nel tipo.

Esempio: La figura seguente mostra l'area di memoria allocata per la variabile persona_1.

persona_1

nome	cognome	giorno	mese	anno
(4 byte)	(4 byte)	(2 byte)	(2 byte)	(2 byte)

Come si può vedere, la memoria allocata ha una struttura simile a quella di un array, con la differenza che i campi possono essere di dimensione diversa.

7.1.2. Creare nuovi tipi di dato record con struct

Sebbene il C permetta di dichiarare variabili di tipo record specificandone contestualmente la struttura (come visto sopra), questa pratica è sconsigliata in quanto, nel caso di più dichiarazioni di variabile in diversi punti del programma, richiede che la stessa struttura sia ripetuta più volte, con conseguenze negative anche sulla manutenibilità del programma. Inoltre, e forse questo è l'aspetto più importante, variabili dichiarate in punti diversi sono considerate di tipi diversi, anche nel caso in cui i rispettivi record siano identici.

Esempio: Il seguente frammento di codice contiene dichiarazioni di variabile con record aventi strutture identiche, in punti diversi del programma.

```
int main (int argc, char** argv){
//...
    struct{
        char* nome;
        char* cognome;
        short int giorno;
        short int mese;
        short int anno;
    } persona_1, persona_2;
//...
    struct{
        char* nome;
        char* cognome;
        short int giorno;
        short int mese;
        short int anno;
    } persona_3;
}
```

Se dopo aver realizzato il programma si dovessero apportare modifiche alla struttura del record, ad esempio aggiungere un campo, queste andrebbero effettuate in entrambe le dichiarazioni. Si noti inoltre che le variabili persona_1 e persona_2 sono dello stesso tipo, ma la variabile persona_3 è considerata di un altro tipo, nonostante l'uguaglianza nella struttura dei record.

7.1.3. Tag di struttura

Un primo modo per identificare un record è tramite i cosiddetti *tag di struttura*, ovvero etichette che identificano la struttura di un record.

Dichiarazione di un tag di struttura

Sintassi:

```
struct nome-tag {
    dichiarazione-variabile-1;
    ...
    dichiarazione-variabile-n;
};
```

dove:

- *nome-tag* è l'etichetta che si vuole usare per identificare la struttura;

Semantica: Viene definito il tag *nome-tag*.

Una volta definiti, i tag di struttura possono essere usati per definire variabili.

Esempio: Nel seguente frammento di codice, viene definito il tag di struttura persona, successivamente utilizzato nelle dichiarazioni delle variabili persona_1, persona_2 e persona_3.

persona-tag.h

```
// File persona-tag.h
#ifndef PERSONATAG_H
#define PERSONATAG_H

// Definizione tag struttura
struct persona{
    char *nome, *cognome;
    short int giorno, mese, anno;
};

#endif
```

persona-tag.c

```c
// File persona-tag.c

#include "persona-tag.h"

int main(int argc, char** argv){
    //...
    struct persona persona_1, persona_2;
    //...
    struct persona persona_3;
}
```

Per definire variabili di tipo record mediante tag di struttura, è necessario l'uso della parola chiave struct prima del tag di struttura, come mostrato nell'esempio. Variabili definite in questo modo hanno tipi compatibili. Si noti la strutturazione del codice in più file: nel file header (persona-tag.h) viene definito il tag di struttura, successivamente usato nel file persona-tag.c.

7.1.4. Definire tipi di dato record con typedef

Il secondo modo messo a disposizione dal C per identificare una struttura con un nome simbolico consiste nel definire un tipo record.

Definizione di un nuovo tipo di dato record

Sintassi:

```
typedef struct {
    dichiarazione-variabile-1;
    ...
    dichiarazione-variabile-n;
} nome;
```

dove:

- *nome* è il nome del nuovo tipo che si vuole definire;
- *dichiarazione-variabile-i* è la dichiarazione dell'*i*-esimo campo del nuovo tipo;

Semantica:

Viene definito un nuovo tipo di dato record con nome *nome*, avente come membri i campi in esso dichiarati, ciascuno del rispettivo tipo.

Si noti la presenza del carattere ;, sempre obbligatoria, al termine della definizione. Come già discusso, i nuovi tipi di dato vengono di norma definiti nei file header, nella sezione relativa alle dichiarazioni di tipo, dopo le direttive di inclusione e prima di dichiarare eventuali funzioni.

Esempio: Il seguente frammento di codice mostra una possibile definizione del record Persona.

persona.h

```
// File persona.h
#ifndef PERSONA_H
#define PERSONA_H

typedef struct{
    char* nome;
    char* cognome;
    short int giorno;
    short int mese;
    short int anno;
} Persona;

#endif
```

Questa dichiarazione sta di fatto introducendo un nuovo tipo di dato, Persona, che sarà utilizzabile come ogni altro tipo.

Si può anche definire un nuovo tipo di dato record combinando typedef e tag di struttura.

Esempio: Nel seguente file, il tag di struttura persona viene usato per definire il tipo record Persona.

```
//Definizione tag:
struct persona{
    char* nome;
    char* cognome;
    short int giorno;
    short int mese;
    short int anno;
};

// Definizione tipo
typedef struct persona Persona;
```

Questa dichiarazione di tipo è equivalente alla precedente.

La definizione di un nuovo tipo per identificare un tipo di dato record è la soluzione da preferire. Salvo casi particolari, questo è il modo in cui saranno definiti i tipi di dato record nel seguito.

In C++ è possibile definire un nuovo tipo record direttamente con la definizione struct.

Nell'esempio precedente, in C++, persona è già un tipo record e può essere usato senza dover definire un nuovo tipo Persona tramite typedef.

7.1.5. Variabili di tipo record

Una volta definito un tipo di dato record, esso, come ogni tipo definito dall'utente, diventa disponibile per l'uso.

Esempio: Nel seguente programma viene dichiarata la variabile persona_1 di tipo Persona

```
#include "persona.h"
//...
int main(int argc, char** argv){
  Persona persona_1;
}
```

Si osservi che la variabile persona_1 è dichiarata come una comune variabile, trattando Persona come un comune tipo di dato. In particolare, non è richiesto l'uso della parola chiave struct nella dichiarazione di variabile. Chiaramente, variabili distinte di tipo Persona sono dello stesso tipo.

7.1.6. Accesso ai campi di un record

Per accedere al campo di un record si utilizza l'operatore . (punto). I campi di un record possono essere trattati, in maniera simile a quanto avviene per gli array, come variabili.

Esempio: Il seguente programma dichiara la variabile p1 di tipo Persona, ne inizializza i campi e successivamente li stampa.

persona.c

```
#include <stdio.h>
#include "persona.h"

int main(int argc, char** argv){

    Persona p1;
    p1.nome = "Mario";
    p1.cognome = "Rossi";
    p1.giorno = 7;
    p1.mese = 4;
    p1.anno = 1964;

    printf("%s %s, %d/%d/%d\n", p1.nome, p1.cognome,
           p1.giorno, p1.mese, p1.anno);
}
```

7.1.7. Inizializzazione di variabili di tipo record

Un'alternativa all'inizializzazione dei campi di un record uno ad uno è l'utilizzo delle parentesi, in maniera simile a quanto visto per gli array. Questa può avvenire su base posizionale o tramite *designatori*, overo espressioni del tipo *.campo-i*.

Esempio: La seguente linea di codice dichiara ed inizializza la variabile p1 di tipo Persona, associando l'*i*-esimo valore all'*i*-esimo campo, secondo l'ordine di comparizione nella definizione del tipo:

```
Persona p1 = {"Mario","Rossi",7,4,1964};
```

La seguente linea di codice dichiara ed inizializza la variabile p1 di tipo Persona assegnando esplicitamente a ciascun campo un valore:

```
Persona p1 = {
    .cognome="Rossi",
    .giorno = 7,
    .mese = 4,
    .nome="Mario",
    .anno = 1964
};
```

In questo caso, l'ordine con cui i campi vengono assegnati non conta, in quanto il campo a cui assegnare il valore è identificato dal designatore.

Entrambe le inizializzazioni degli esempi appena visti hanno esattamente lo stesso effetto della dichiarazione e delle assegnazioni campo a campo viste in precedenza.

L'inizializzazione di record può aver luogo solo contestualmente alla sua dichiarazione.

7.1.8. Assegnazione e uguaglianza

7.1.8.1. Assegnazione di record

Per quanto riguarda l'assegnazione, le variabili di tipo record vengono trattate allo stesso modo delle variabili di tipi primitivi. In particolare, è possibile assegnare ad una variabile di tipo record un valore di tipo record di tipo compatibile con quello della variabile.

Esempio:

```
Persona p1 = ... ;
Persona p2;
p2 = p1;
```

L'effetto dell'assegnazione è la copia campo a campo del record restituito dall'espressione p1, cioè il contenuto della variabile, nel record contenuto nella variabile p2. Si osservi la differenza rispetto agli array, per i quali l'assegnazione è disponibile solo considerando gli array come puntatori e avviene quindi con condivisione di memoria.

Nella copia campo a campo, anche eventuali campi di tipo array *allocati staticamente* (ad esempio char[256]) all'interno del record vengono copiati. Gli array allocati dinamicamente (ad esempio char *) invece vengono copiati come puntatori e quindi comportano una condivisione di memoria.

Se si considera che l'assegnazione non è un'operazione disponibile su variabili di tipo array, questo comportamento potrebbe apparire sorprendente. Tuttavia, si osservi che il nome di una variabile di tipo struttura identifica una variabile. Quindi, poiché l'assegnazione tra variabili, ad esempio p2 = p1, prevede la copia del contenuto della memoria di una variabile nella memoria dell'altra, se la prima variabile contiene un array allocato staticamente come membro, esso viene automaticamente copiato per effetto dell'assegnazione tra le variabili di tipo record. Una variabile di un array invece identifica l'indirizzo di memoria dell'array e pertanto l'assegnazione opera su puntatori. In un certo senso, l'uso di una struttura rende disponibile, indirettamente, l'assegnazione tra array allocati staticamente.

Esempio: Si consideri il seguente programma. Il record PuntoColorato, definito nel file punto-colorato.h rappresenta un punto nello spazio cartesiano, a cui è associato un colore. Nello stesso file è inoltre dichiarata la funzione stampaPuntoColorato che stampa su schermo il contenuto del parametro p di tipo PuntoColorato. La definizione di tale funzione è fornita nel file punto-colorato.c. Il file punto-colorato-main.c mostra invece un esempio d'uso del record e della funzione dichiarati nel file header.

punto-colorato.h

```
// File punto-colorato.h
#ifndef PUNTOCOLORATO_H
#define PUNTOCOLORATO_H

typedef struct{
    float coord[3]; // coordinate nello spazio
    char colore[256]; // colore del punto
} PuntoColorato;

void stampaPuntoColorato(PuntoColorato p);
#endif
```

punto-colorato.c

```
// File punto-colorato.c

#include <stdio.h>
#include "punto-colorato.h"

void stampaPuntoColorato(PuntoColorato p){
```

```
    printf("((%f,%f,%f),%s)\n",
           p.coord[0], p.coord[1], p.coord[2],
           p.colore);
}
```

punto-colorato-main.c

```
// File punto-colorato-main.c
#include <stdio.h>
#include "punto-colorato.h"

int main(int argc, char** argv){
    PuntoColorato pc1 = {{0,0,0},"bianco"};
    PuntoColorato pc2 = pc1;

    stampaPuntoColorato(pc1);
    stampaPuntoColorato(pc2);

    pc1.coord[0] = 3; // coord di pc1 cambia

    stampaPuntoColorato(pc2); // pc2 non cambia
}
```

Come detto, l'effetto dell'assegnazione è quello di copiare campo a campo il record p1 nel record p2. La seguente figura mostra lo stato della memoria immediatamente dopo l'esecuzione della linea PuntoColorato pc2 = pc1;

pc1

coord			colore
0	0	0	"bianco"

pc2

coord			colore
0	0	0	"bianco"

Si osservi che ciascun record contiene un proprio array distinto dall'altro. Pertanto, dopo l'esecuzione, nel programma principale, della linea pc1.coord[0] = 3; , l'array contenuto in pc1 sarà cambiato, mentre quello in pc2 resterà invariato:

pc1

coord			colore
3.0	0	0	"bianco"

pc2

coord			colore
0	0	0	"bianco"

Si noti che in questo esempio il campo colore del record PuntoColorato è dichiarato in maniera statica char[256] e viene quindi copiato

dall'istruzione di assegnazione. Si ricorda anche che un array allocato staticamente non può essere assegnato con l'istruzione di assegnazione in seguito alla sua dichiarazione, quindi nel codice mostrato in precedenza l'istruzione pc1.colore = "bianco" provocherebbe un errore a tempo di compilazione, in quanto la stringa "bianco" è di tipo const char [7] e l'istruzione sarebbe intepretata come assegnazione tra array statici non consentita in C. Per accedere in scrittura all'area di memoria del campo colore si può usare ad esempio la funzione strcpy.

7.1.8.2. Uguaglianza tra record

Sebbene il C metta a disposizione l'operatore di assegnazione, esso non permette il test uguaglianza tra record. In altre parole, gli operatori == e != non sono applicabili a record.

7.1.9. Puntatori a record

È anche possibile dichiarare puntatori a variabili di tipo record.

Esempio: Il seguente frammento di codice dichiara una variabile p di tipo puntatore a Persona

```
Persona* p;
```

Per accedere ai campi di un record tramite il puntatore, una prima alternativa consiste nell'accedere prima alla variabile puntata dal puntatore, mediante l'operatore *, e successivamente accedere al campo d'interesse.

Esempio: Nel seguente frammento di codice, l'accesso al campo nome della variabile p di tipo Persona viene effettuato tramite il puntatore punt_p a Persona.

```
Persona p = {"Mario","Rossi",7,4,1964};
Persona *punt_p = &p;
printf("Nome: %s\n",(*punt_p).nome);
```

Si osservi l'uso delle parentesi nell'espressione (*punt_p).nome, necessarie in quanto l'operatore . ha precedenza più alta rispetto all'operatore *.

La seconda alternativa, più comoda e comunemente usata, consiste nell'uso dell'operatore -> (freccia a destra).

Esempio: Si consideri il seguente frammento di codice

```
Persona p = {"Mario","Rossi",7,4,1964};
Persona *punt_p = &p;
```

Le seguenti espressioni sono equivalenti:

```
(*punt_p).nome
punt_p->nome
```

Pertanto, entrambe le seguenti istruzioni stampano la stringa Mario

```
printf("%s\n", (*punt_p).nome);
printf("%s\n", punt_p->nome);
```

Passaggio di parametri e restituzione di un risultato di tipo record da parte di funzioni avvengono secondo gli stessi meccanismi visti per le variabili di tipi primitivi.

Esempio: La seguente funzione prende in input un parametro di tipo Persona ed un intero che rappresenta un anno e restituisce l'età della persona, calcolata come differenza tra l'anno corrente e l'anno di nascita della persona.

```
#include <stdio.h>
#include "persona.h"

int calcolaEta(Persona p, int anno_corrente){
    return anno_corrente - p.anno;
}

// Esempio d'uso:

int main(){
    Persona p = {"Mario","Rossi",7,4,1964};
    printf("eta': %d\n",calcolaEta(p,2018));
}
```

Diversamente da quanto accade per gli array, il passaggio di parametri di tipo record avviene *per valore*: al momento dell'invocazione della funzione, il record viene copiato nel RDA della funzione.

Esempio: La seguente funzione prende in input un parametro di tipo Persona e vi apporta delle modifiche. Tuttavia, grazie al meccanismo di passaggio per valore, tali modifiche non sono visibili al modulo chiamante.

```
#include <stdio.h>
#include "persona.h"

void cambiaEta(Persona p, int nuovo_anno){
    p.anno = nuovo_anno;
}

// Esempio d'uso:

int main(){
    Persona p = {"Mario","Rossi",7,4,1964};
    cambiaEta(p,2000);
    printf("anno: %d\n",p.anno); // stampa 1964
}
```

Anche il caso in cui una funzione prende in input un puntatore a record è sostanzialmente analogo al caso di variabili di tipi primitivi. In particolare, tramite puntatori è possibile effettuare side-effect sul record puntato dal parametro, in maniera tale da rendere visibili al modulo chiamante le modifiche apportate.

Esempio: La seguente funzione prende in input un puntatore a record di tipo Persona e ne assegna i campi a valori di default. Poiché effettuate tramite puntatore, le modiche effettuate dalla funzione sono visibili nel programma principale, dopo la sua invocazione.

```
#include <stdio.h>
#include "persona.h"

void inizializzaPersona(Persona* p){
    p -> nome = "";
    p -> cognome = "";
    p -> giorno = 0;
    p -> mese = 0;
    p -> anno = 0;
}

// Esempio d'uso:

int main(int argc, char** argv){
    Persona p1;
    inizializzaPersona(&p1);
    printf("%s %s, %d/%d/%d\n", p1.nome, p1.cognome,
           p1.giorno, p1.mese, p1.anno);
}
```

Al pari dei valori di tipi primitivi, una funzione può restituire un record.

Esempio: La seguente funzione assegna alla variabile locale p di tipo Persona un record i cui campi sono popolati con il valore dei parametri. Il contenuto della variabile viene quindi restituito.

```
#include <stdio.h>
#include "persona.h"

Persona nuovaPersona(char* nome, char* cognome, int g,
     int m, int a){
    Persona p;
    p.nome = nome;
    p.cognome = cognome;
    p.giorno = g;
    p.mese = m;
    p.anno = a;
  return p;
}

// Esempio d'uso:

int main(int argc, char** argv){
    Persona p1 = nuovaPersona("Mario", "Rossi",
    10,2,1990);
    printf("%s %s, %d/%d/%d\n", p1.nome, p1.cognome,
           p1.giorno, p1.mese, p1.anno);
}
```

Si osservi che la funzione restituisce una copia del valore memorizzato nella variabile p che viene assegnato, nella funzione main, alla variabile p1. Essendo il valore un record, l'assegnazione avviene, come visto, campo a campo.

7.1.10. Allocazione dinamica e deallocazione di record

La funzione malloc può essere usata per allocare dinamicamente variabili di tipo record.

```
Persona *p1 = (Persona*) malloc(sizeof(Persona));
```

Per i campi di tipo puntatore (ad esempio, array dichiarati dinamicamente, come char *) non viene allocata memoria. Pertanto è necessario allocare (e in seguito deallocare) esplicitamente la memoria per tutti i campi di tipo puntatore.

Per il resto, non vi sono sostanziali differenze con il caso di variabili di tipi primitivi, eccetto che, una volta allocata una variabile di tipo record, per accedere ai suoi membri occorre utilizzare l'operatore -> (o gli operatori * e . opportunamente combinati).

Anche la deallocazione viene eseguita in maniera analoga al caso di variabili di tipi primitivi, ovvero tramite la funzione free:

```
free(p1);
```

Nel caso in cui il record contiene campi di tipo puntatore allocati dinamicamente, è necessario rilasciare la memoria associata a tali campi, prima di rilasciare la memoria associata al record.

Esempio: Il seguente programma mostra la dichiarazione dinamica di una variabile di tipo Persona e l'accesso ai suoi campi.

```
#include <stdio.h>
#include <stdlib.h>
#include "persona.h"

int main(int argc, char** argv){
    Persona* p1 = (Persona*) malloc(sizeof(Persona));
    p1 -> nome = "Mario";
    p1 -> cognome = "Rossi";
    p1 -> giorno = 7;
    p1 -> mese = 4;
    p1 -> anno = 1964;

    printf("%s %s, %d/%d/%d\n", p1->nome, p1->cognome,
           p1->giorno, p1->mese, p1->anno);

    free(p1);
}
```

Esempio: La seguente funzione alloca dinamicamente un record di tipo Persona e ne restituisce il puntatore. Nella funzione main, il record viene utilizzato e, quando non piú necessario, deallocato.

```
#include <stdio.h>
#include <stdlib.h>
#include "persona.h"

Persona* puntNuovaPersona(char* nome, char* cognome,
    int g, int m, int a){
  Persona* p = (Persona*) malloc(sizeof(Persona));
    p -> nome = nome;
    p -> cognome = cognome;
    p -> giorno = g;
    p -> mese = m;
    p -> anno = a;
  return p;
}

// Esempio d'uso:

int main(int argc, char** argv){
    Persona* p1 = puntNuovaPersona("Mario", "Rossi",
    10,2,1990);
    printf("%s %s, %d/%d/%d\n", p1 -> nome, p1 ->
    cognome, p1 -> giorno, p1 -> mese, p1 -> anno);
  // ...
  free(p1);
}
```

7.1.11. Record per la rappresentazione di matrici

I record sono particolarmente utili per definire un nuovo tipo di dato per memorizzare matrici allocate dinamicamente.

Esempio: dichiarazione di un tipo Mat per rappresentare matrici di float.

```
typedef struct {
  int rows;
  int cols;
  float **row_ptrs;
} Mat;
```

Nella definizione precedente, Mat è un nuovo tipo che contiene tre campi: rows: numero di righe della matrice, cols: numero di colonne della matrice, row_ptrs: puntatore all'array di array di float che contiene i valori della matrice.

L'accesso agli elementi della matrice avviene tramite i campi del record.

```
Mat *m = ...;  // allocazione dinamica della matrice

for (int i=0; i<m->rows; i++) {
  for (int j=0; j<m->cols; j++) {
     ... m->row_ptrs[i][j]  // operazione su elemento
    i,j
  }
}
```

7.1.12. Record annidati

Come detto, una volta definito un tipo record, esso può essere usato come un qualsiasi altro tipo, ad esempio come campo di un record.

Esempio: Nel seguente file il tipo Punto viene sfruttato per fornire una definizione alternativa del tipo PuntoColorato.

punto-colorato2.h

```
#ifndef PUNTOCOLORATO2_H
#define PUNTOCOLORATO2_H

typedef struct{
    float X;
```

```
    float Y;
    float Z;
} Punto;

typedef struct{
    Punto punto;
    char colore[256];
} PuntoColorato2;

#endif
```

Un esempio d'uso del tipo PuntoColorato2 è il seguente:

punto-colorato2-main.c

```
#include <stdio.h>
#include "punto-colorato2.h"

int main (int argc, char** argv){
    Punto p = {2.0f,2.2f,3.5f};
    PuntoColorato2 pc = {p,"verde"};

    printf("((%f,%f,%f),%s)\n",
        pc.punto.X,pc.punto.Y,pc.punto.Z,pc.colore);
}
```

Si osservi l'uso dell'operatore . per accedere prima al campo punto di pc (ottenuto con l'espressione pc.punto) e successivamente per accedere ai campi di punto, ad esempio pc.punto.X per ottenerne la coordinata X.

L'unica limitazione alla struttura di un record consiste nel fatto che quando si definisce un tipo record T, la sua struttura non può menzionare il tipo T al suo interno. Questa restrizione può tuttavia essere superata tramite l'uso di tag (v. seguito).

7.1.13. Record autoreferenziali

Un caso di record di particolare interesse è quello in cui si vuole che un record di tipo T faccia riferimento, tramite un suo campo, ad un altro record dello stesso tipo. Sebbene questo non introduca nessuna novità da un punto di vista concettuale, la limitazione sopra discussa impone l'uso di tag.

Esempio: Nel seguente frammento di codice viene definito un record con tag di struttura parente, utilizzato all'interno della struttura stessa. Il tag viene successivamente usato per definire il tipo Parente.

parente.h

```
#ifndef PARENTE_H
#define PARENTE_H

// Tag di struttura:
struct parente{
    char nome[256];
    char cognome[256];
    struct parente* padre;
    struct parente* madre;
};

// Definizione di tipo:
typedef struct parente Parente;
#endif
```

Si osservi che all'interno della definizione del record non si possono dichiarare campi di tipo struct parente, ad esempio padre, ma solo puntatori ad essi. Questo è dovuto al fatto che, nel momento in cui viene dichiarato il campo, il record parente è ancora in corso di definizione, ovvero è *incompleto*. Mentre il C vieta la dichiarazione di variabili di tipo incompleto, permette la dichiarazione di puntatori a tali variabili.

Faremo ampiamente uso di record di questo tipo quando tratteremo le *strutture collegate*.

7.1.14. Array di record

Infine, notiamo che è anche possibile dichiarare array di record.

Esempio: Il seguente programma dichiara un array famiglia di record di tipo Persona e ne inizializza la prima componente.

```c
#include <stdio.h>
#include "persona.h"

int main (int argc, char** argv){
    Persona famiglia[3];

    famiglia[0].nome = "Mario";
    famiglia[0].cognome = "Rossi";
    famiglia[0].giorno = 7;
    famiglia[0].mese = 4;
    famiglia[0].anno = 1964;

    //...
}
```

Si noti che l'accesso alle componenti di ciascun record si effettua accedendo dapprima al record tramite l'operatore [] e successivamente usando il punto. Ad esempio l'espressione famiglia[0].nome denota il campo nome del record memorizzato come prima componente dell'array famiglia, ovvero famiglia[0].

7.2. Unioni

Un ulteriore tipo di dato strutturato messo a disposizione del C è union, o unione. union è un tipo che si comporta in maniera simile ad un record ma, a differenza di questo, *memorizza solo un campo alla volta*.

Variabili di tipo union e tipi union vengono rispettivamente dichiarate e definiti allo stesso modo dei record, sostituendo la parola chiave struct con union.

Esempio: Il seguente frammento di codice definisce il tipo di dato chardouble come una unione contenente i campi c di tipo char e d di tipo double.

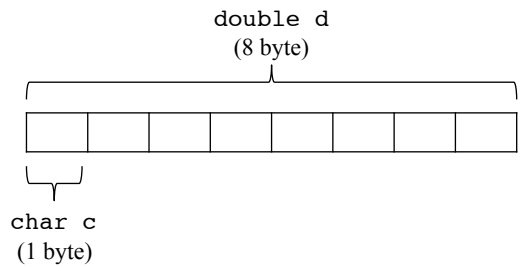
```c
#ifndef CHARDOUBLE_H
#define CHARDOUBLE_H

typedef union{
    char c;
    double d;
} chardouble;

#endif
```

Una volta definito, il tipo di dato può essere usato per dichiarare variabili.

Esempio:

```
chardouble cd;
```

La dichiarazione di variabile ha per effetto l'allocazione di uno spazio di memoria di dimensione pari alla *dimensione massima tra quelle dei tipi dei campi della union*. Ad esempio, la dimensione della variabile cd appena definita è pari a sizeof(double) byte (tipicamente 8). La figura seguente mostra l'organizzazione della memoria associata alla variabile cd:



La caratteristica distintiva dei tipi union consiste nel fatto che *solo l'ultimo campo a cui è stato assegnato un valore è significativo*, ovvero contiene effettivamente il valore ad esso assegnato, mentre il valore contenuto negli altri campi è soggetto a modifiche arbitrarie. Questo deriva dal fatto che diversi campi condividono una stessa regione di memoria (ad es. il primo byte nella figura precedente) e quindi le modifiche apportate ad ognuno di essi hanno un impatto anche sugli altri. Per capire come ciò avvenga, si consideri nuovamente l'organizzazione della memoria della variabile cd. Come si può vedere i due campi c e d della variabile condividono il primo byte, pertanto ogni assegnazione al campo c avrà un impatto sul valore corrente del campo d e viceversa.

L'accesso ai campi di una union si effettua con le stesse modalità del caso dei record. Ad esempio, l'espressione cd.c denota il campo c della variabile c.

Esempio: Il seguente frammento di codice mostra l'uso di una variabile di tipo chardouble.

```
chardouble cd;
cd.c = 'a';
/* Solo il campo c e' significativo
(fino alla prossima assegnazione)*/
// ...
cd.d = 30.0;
/*Solo il campo d e' significativo
(fino alla prossima assegnazione)*/
//...
```

Il tipo di dato union è tipicamente usato per dichiarare variabili che possono assumere valori di diversi tipi. Ad esempio, la variabile cd può assumere, attraverso l'assegnazione ai suoi campi, valori di tipo char o double. Notiamo che un comportamento simile potrebbe essere ottenuto definendo un record, ad esempio:

```
typedef struct {
  char c;
  double d;
} doublechar;
```

quindi dichiarando una variabile doublechar dc, ed accedendo di volta in volta solo all'ultimo campo a cui è stato assegnato un valore. Si osservi tuttavia che questo approccio richiede una maggiore quantità di memoria.

Ad eccezione di questa importante differenza, le variabili di tipo union vengono trattate in C allo stesso modo dei record, in particolare per quanto riguarda:

- definizione di tipi e dichiarazione di variabili;
- passaggio di parametri (che avviene per valore);
- restituzione di valori di tipo union;
- accesso tramite puntatori (operatore ->).

7.3. Tipi di dato enumerati

In C è possibile definire tipi di dato che consistono in un insieme di valori enumerati esplicitamente.

Definizione di un nuovo tipo di dato enumerato

Sintassi:

```
typedef enum {
    valore-1,
    ...
    valore-n
} nome;
```

dove:

- *nome* è il nome del nuovo tipo che si vuole definire;
- *valore-i* è l'*i*-esimo elemento del tipo;

Semantica: Viene creato un nuovo tipo di dato enumerato i cui valori sono tutti e solo quelli elencati nella dichiarazione.

Dopo essere stato dichiarato, un tipo enumerato può essere usato come ogni altro tipo.

Esempio: Il seguente file header dichiara un tipo enumerato usato per memorizzare una direzione (alto, basso, destra, sinistra).

direzione.h

```
// File direzione.h

#ifndef DIREZIONE_H
#define DIREZIONE_H

typedef enum{
    ALTO,
    BASSO,
    DESTRA,
    SINISTRA
} Direzione;

#endif
```

Il tipo può essere usato per definire una variabile di tipo Direzione:

```
Direzione d;
```

La variabile può essere usata come una qualunque altra variabile:

```
d = BASSO;
//...
if (d == ALTO) {...}
if (d == BASSO) {...}
//...
```