

Tipi di dato utente

Laboratorio di Programmazione I

Corso di Laurea in Informatica
A.A. 2017/2018



Calendario delle lezioni

Ogni lezione consta di una spiegazione assistita da slide, e seguita da esercizi in classe

Lezione 1 (19-20/09/2017) - Introduzione all'ambiente Linux

Lezione 2 (26-27/09/2017) - Introduzione al C

Lezione 3 (3-4/10/2017) - Tipi primitivi e costrutti condizionali

Lezione 4 (10-11/10/2017) - Costrutti iterativi ed array

Lezione 5 (17-18/10/2017) - Funzioni, stack e visibilità variabili

Lezione 6 (24-25/10/2017) - Puntatori e Aritmetica dei
puntatori

Lezione 7 (7-8/11/2017) - Tipi di dati utente

Lezione 8 (14-15/11/2017) - Liste concatenate e librerie.

Lezione 9 (21-22/11/2017) - Debugging

Lezione 10 (28-29/11/2017) - Ricorsione

Lezione 11 (5-6/12/2017) - CAML

Lezione 12 (12-13/12/2017) - Simulazione prova pratica

Sommario

- 1 Tipi di dato personalizzati
 - struct
 - enum e typedef
- 2 Array di struct

Outline

- 1 Tipi di dato personalizzati
 - struct
 - enum e typedef
- 2 Array di struct

Sommario

- 1 Tipi di dato personalizzati
 - struct
 - enum e typedef
- 2 Array di struct

Definizione di una `struct`

- Array: N oggetti tutti dello **stesso tipo**.
- **struct**: tipo nuovo, definito dal programmatore
 - costituito da variabili che hanno tipi di dato primitivi
 - può contenere array
 - può contenere altre struct (ma ci sono delle regole)

Definizione di una `struct`

- Array: N oggetti tutti dello **stesso tipo**.
- **struct**: tipo nuovo, definito dal programmatore
 - costituito da variabili che hanno tipi di dato primitivi
 - può contenere array
 - può contenere altre struct (ma ci sono delle regole)

Sintassi:

```
struct nome_struct {  
    tipo1 nome1; tipo2 nome2; ...  
};
```

Definizione di una `struct`

- Array: N oggetti tutti dello **stesso tipo**.
- **struct**: tipo nuovo, definito dal programmatore
 - costituito da variabili che hanno tipi di dato primitivi
 - può contenere array
 - può contenere altre struct (ma ci sono delle regole)

Sintassi:

```
struct nome_struct {  
    tipo1 nome1; tipo2 nome2; ...  
};
```

Esempio:

```
struct gatto {  
    int eta;           // eta '(anni felini)  
    double peso;      // peso(Kg)  
    double cibo[7];   // cibo assunto nell'ultima settimana(Kg)  
};
```


Utilizzo struct

- 1 La struct va definita all'esterno delle funzioni.

Utilizzo struct

- 1 La struct va definita all'esterno delle funzioni.
- 2 Per accedere agli elementi della **struct**, sia in lettura che in scrittura, si utilizza la seguente sintassi:

```
nome_struct.nome_campo
```

Utilizzo struct

```
struct gatto {  
    int eta;           // eta' (anni felini)  
    double peso;      // peso (Kg)  
    double cibo[7];   // cibo assunto nell'ultima  
                      settimana (Kg)  
};  
  
int main(){  
    struct gatto felix;  
    struct gatto luna;  
  
    felix.eta = 34;  
    felix.peso = 6.5;  
    luna.eta = felix.eta - 5;  
    luna.cibo[1] = 14.5;  
    luna.peso = 15.3;  
  
    if (luna.peso > 10)  
        printf("Luna ha mangiato troppe crocchette.\n");  
}
```

Inizializzazione `struct`

I valori delle variabili contenute in una `struct` possono essere inizializzati con la seguente sintassi. I valori vengono assegnati alle variabili nello stesso ordine in cui compaiono nella definizione della `struct`.

```
struct gatto {  
    int eta;           // eta' (anni felini)  
    double peso;      // peso (Kg)  
    double cibo[7];  // cibo assunto nell'ultima  
                    settimana (Kg)  
};  
  
int main() {  
    struct gatto felix = {14, 5.7, {0.2, 0.1, 0.3, 0.3,  
    0.2, 0.5, 0.3}};  
    return 0;  
}
```

Passaggio di una struct per parametro

Supponiamo di avere una **struct** gatto di nome felix:

```
struct gatto felix;
```

Passaggio di una struct per parametro

Supponiamo di avere una **struct** gatto di nome `felix`:

```
struct gatto felix;
```

Il passaggio delle **struct** per parametro funziona allo stesso modo del passaggio delle variabili:

- Se volete creare una **copia** delle variabili di `felix` nel frame della funzione, passate il valore della **struct**:
`controlla_peso(felix);`

Passaggio di una struct per parametro

Supponiamo di avere una **struct** gatto di nome `felix`:

```
struct gatto felix;
```

Il passaggio delle **struct** per parametro funziona allo stesso modo del passaggio delle variabili:

- Se volete creare una **copia** delle variabili di `felix` nel frame della funzione, passate il valore della **struct**:
`controlla_peso(felix);`
- Se invece volete passare un **riferimento** alle variabili di `felix`, in modo che le modifiche siano visibili anche da fuori, passate l'indirizzo della **struct**:
`aggiorna_peso(&felix, nuovo_peso);`

Accesso alle struct

```
struct gatto {
    int eta;           // eta' (anni felini)
    double peso;      // peso (Kg)
    double cibo[7];   // cibo assunto nell'ultima
    settimana (Kg)
};

void aggiorna_peso(struct gatto* gatto, double peso){
    (*gatto).peso = peso;
};

int main(){
    struct gatto felix = {14, 5.7, {0.2, 0.1, 0.3,
    0.3, 0.2, 0.5, 0.3}};
    aggiorna_peso(&felix, 6.2);
    return 0;
}
```


Accesso alle struct

```
struct gatto {
    int eta;           // eta' (anni felini)
    double peso;      // peso (Kg)
    double cibo[7];   // cibo assunto nell'ultima
    settimana (Kg)
};

void aggiorna_peso(struct gatto* gatto, double peso){
    gatto->peso = peso;
};

int main(){
    struct gatto felix = {14, 5.7, {0.2, 0.1, 0.3,
    0.3, 0.2, 0.5, 0.3}};
    aggiorna_peso(&felix, 6.2);
    return 0;
}
```

Accesso alle `struct`

- Per accedere alle variabili di una `struct` `s` di cui abbiamo il *valore*

```
struct nome_struct s
s.nome_var
```

- Per accedere alle variabili di una `struct` `s` di cui abbiamo l'*indirizzo*

```
struct nome_struct* s
s->nome_var
```

oppure

```
(*s).nome_var
```

struct e array

Una **struct** può contenere un array, che si può accedere nello stesso modo in cui si accedono i tipi primitivi.

struct e array

```
struct gatto {
    int eta;           // eta' (anni felini)
    double peso;      // peso (Kg)
    double cibo[3];   // cibo assunto negli ultimi
    tre giorni (Kg)
};
int main(){
    struct gatto felix;
    int i;
    double somma_cibo = 0;

    felix.eta = 34;
    felix.cibo[0] = 0.50;
    felix.cibo[1] = 0.22;
    felix.cibo[2] = 0.56;

    for (i=0; i<3; i++)
        somma_cibo += felix.cibo[i];
    printf("In tre giorni Felix ha mangiato %f Kg.",
    somma_cibo);
}
```

Dimensione struct

```
struct gatto {  
    int eta;           // eta' (anni felini)  
    double peso;      // peso (Kg)  
    double cibo[7];   // cibo assunto nell'ultima  
                      settimana (Kg)  
};
```

La dimensione di una **struct** è maggiore o uguale la somma delle dimensioni dei suoi tipi primitivi.

Dimensione struct

```
struct gatto {  
    int eta;           // eta' (anni felini)  
    double peso;      // peso (Kg)  
    double cibo[7];   // cibo assunto nell'ultima  
                      settimana (Kg)  
};
```

La dimensione di una **struct** è maggiore o uguale la somma delle dimensioni dei suoi tipi primitivi.

```
sizeof(struct gatto) >=  
sizeof(int)+sizeof(double)+7*sizeof(double)
```

Per semplicità assumiamo l'uguaglianza, tuttavia usate sempre `sizeof(struct)` anziché la somma delle `sizeof(..)` dei suoi componenti.

struct annidate

- Una **struct** può contenere un'altra **struct**, ma ci sono delle regole.
- La dimensione della **struct** deve essere finita, costante e determinabile a tempo di compilazione, quindi non può esserci ricorsione!
- Una struct S_2 può contenere una **struct** S_1 solo se S_1 è definita prima nel programma.
- In questo modo l'annidamento delle **struct** forma un **albero**.

struct annidate

```
struct citta {  
    int n_cittadini;  
    double latitudine;  
    double longitudine;  
};  
struct nazione {  
    struct citta capitale;  
    double superficie;  
    int fuso_orario;  
};
```


struct annidate

```
struct citta {  
    int n_cittadini;  
    double latitudine;  
    double longitudine;  
};  
struct nazione {  
    struct citta capitale;  
    double superficie;  
    int fuso_orario;  
};
```

La **struct nazione** ha una variabile di tipo **struct citta**.
La **struct citta non può avere** come variabile una **struct nazione**, altrimenti non sarebbe possibile determinare lo spazio occupato dalla **struct**.

struct annidate - errore

```
struct citta {  
    int cittadini;  
    double latitudine;  
    double longitudine;  
    struct nazione n;  
};  
struct nazione {  
    struct citta capitale;  
    double superficie;  
    int fuso_orario;  
};
```

```
sizeof(int) = 4
```

```
sizeof(double) = 8
```

```
sizeof(struct citta) = 4 + 16 + sizeof(struct nazione)
```

```
sizeof(struct nazione) = 4 + 8 + sizeof(struct citta)
```

L'annidamento delle **struct** è ricorsivo, e il compilatore dà **errore**.

struct annidate e puntatori

Eppure ci sono dei casi in cui le struct devono logicamente chiamarsi tra di loro (annidamento ricorsivo). **Esempio:** vogliamo rappresentare un albero genealogico:

```
struct persona {  
    struct persona madre;  
    struct persona padre;  
    int anno_nascita;  
};
```

Avrebbe dimensione “infinita”. Come risolviamo?

struct annidate e puntatori

Eppure ci sono dei casi in cui le struct devono logicamente chiamarsi tra di loro (annidamento ricorsivo). **Esempio:** vogliamo rappresentare un albero genealogico:

```
struct persona {  
    struct persona madre;  
    struct persona padre;  
    int anno_nascita;  
};
```

Avrebbe dimensione “infinita”. Come risolviamo?

I puntatori a **struct** hanno sempre la stessa dimensione (ad es. 8 bytes nelle architetture a 64bit). Soluzione:

struct annidate e puntatori

Eppure ci sono dei casi in cui le struct devono logicamente chiamarsi tra di loro (annidamento ricorsivo). **Esempio:** vogliamo rappresentare un albero genealogico:

```
struct persona {  
    struct persona madre;  
    struct persona padre;  
    int anno_nascita;  
};
```

Avrebbe dimensione “infinita”. Come risolviamo?

I puntatori a **struct** hanno sempre la stessa dimensione (ad es. 8 bytes nelle architetture a 64bit). Soluzione:

```
struct persona {  
    struct persona* madre;  
    struct persona* padre;  
    int anno_nascita;  
};
```

Accesso alle strutture puntate

```
struct persona {  
    struct persona* madre;  
    struct persona* padre;  
    int anno_nascita;  
};  
int main() {  
    struct persona giovanni, alice ;  
    struct persona cecilia;  
    giovanni.anno_nascita = 1980;  
    cecilia.anno_nascita = 2005;  
    cecilia.madre = &alice;  
    cecilia.padre = &giovanni;  
    (cecilia.madre)->anno_nascita = 1982;
```

Accesso alle strutture puntate

```
struct persona {
    struct persona* madre;
    struct persona* padre;
    int anno_nascita;
};
int main() {
    struct persona giovanni, alice ;
    struct persona cecilia;
    giovanni.anno_nascita = 1980;
    cecilia.anno_nascita = 2005;
    cecilia.madre = &alice;
    cecilia.padre = &giovanni;
    (cecilia.madre)->anno_nascita = 1982;
    ((*cecilia.madre)).anno_nascita = 1982; //Equiv.
    alla precedente (a->b e' zucchero sintattico per a (*
    a).b)
    printf("cecilia-madre-anno: %d\n", cecilia.madre->
    anno_nascita);
```

Accesso alle strutture puntate

```
struct persona {
    struct persona* madre;
    struct persona* padre;
    int anno_nascita;
};
int main() {
    struct persona giovanni, alice ;
    struct persona cecilia;
    giovanni.anno_nascita = 1980;
    cecilia.anno_nascita = 2005;
    cecilia.madre = &alice;
    cecilia.padre = &giovanni;
    (cecilia.madre)->anno_nascita = 1982;
    ((*cecilia.madre)).anno_nascita = 1982; //Equiv.
    alla precedente (a->b e' zucchero sintattico per a (*
    a).b)
    printf("cecilia-madre-anno: %d\n", cecilia.madre->
    anno_nascita); //1982
```


Accesso alle strutture puntate

```
struct persona {
    struct persona* madre;
    struct persona* padre;
    int anno_nascita;
};

int main() {
    struct persona giovanni, alice ;
    struct persona cecilia;
    giovanni.anno_nascita = 1980;
    cecilia.anno_nascita = 2005;
    cecilia.madre = &alice;
    cecilia.padre = &giovanni;
    (cecilia.madre)->anno_nascita = 1982;
    ((*cecilia.madre)).anno_nascita = 1982; //Equiv.
    alla precedente (a->b e' zucchero sintattico per a (*
    a).b)
    printf("cecilia-madre-anno: %d\n", cecilia.madre->
    anno_nascita); //1982
    printf("cecilia-padre-anno: %d\n", cecilia.padre->
    anno_nascita);
```

Accesso alle strutture puntate

```
struct persona {
    struct persona* madre;
    struct persona* padre;
    int anno_nascita;
};

int main() {
    struct persona giovanni, alice ;
    struct persona cecilia;
    giovanni.anno_nascita = 1980;
    cecilia.anno_nascita = 2005;
    cecilia.madre = &alice;
    cecilia.padre = &giovanni;
    (cecilia.madre)->anno_nascita = 1982;
    ((*cecilia.madre)).anno_nascita = 1982; //Equiv.
    alla precedente (a->b e' zucchero sintattico per a (*
    a).b)
    printf("cecilia-madre-anno: %d\n", cecilia.madre->
    anno_nascita); //1982
    printf("cecilia-padre-anno: %d\n", cecilia.padre->
    anno_nascita); //1980
```

Accesso alle strutture puntate

```
struct persona {
    struct persona* madre;
    struct persona* padre;
    int anno_nascita;
};

int main() {
    struct persona giovanni, alice ;
    struct persona cecilia;
    giovanni.anno_nascita = 1980;
    cecilia.anno_nascita = 2005;
    cecilia.madre = &alice;
    cecilia.padre = &giovanni;
    (cecilia.madre)->anno_nascita = 1982;
    ((*cecilia.madre)).anno_nascita = 1982; //Equiv.
    alla precedente (a->b e' zucchero sintattico per a (*
    a).b)
    printf("cecilia-madre-anno: %d\n", cecilia.madre->
    anno_nascita); //1982
    printf("cecilia-padre-anno: %d\n", cecilia.padre->
    anno_nascita); //1980
    printf("cecilia: %d\n", cecilia.anno_nascita);
```

Accesso alle strutture puntate

```
struct persona {
    struct persona* madre;
    struct persona* padre;
    int anno_nascita;
};

int main() {
    struct persona giovanni, alice ;
    struct persona cecilia;
    giovanni.anno_nascita = 1980;
    cecilia.anno_nascita = 2005;
    cecilia.madre = &alice;
    cecilia.padre = &giovanni;
    (cecilia.madre)->anno_nascita = 1982;
    ((*cecilia.madre)).anno_nascita = 1982; //Equiv.
    alla precedente (a->b e' zucchero sintattico per a (*
    a).b)
    printf("cecilia-madre-anno: %d\n", cecilia.madre->
    anno_nascita); //1982
    printf("cecilia-padre-anno: %d\n", cecilia.padre->
    anno_nascita); //1980
    printf("cecilia: %d\n", cecilia.anno_nascita); //2005
```

Sommario

- 1 Tipi di dato personalizzati
 - struct
 - enum e typedef
- 2 Array di struct

typedef

typedef: definizione di *alias* per i tipi (nomi alternativi per un tipo di dato già esistente). Esempio:

```
typedef int eta_t;
```

definisce un nuovo tipo `eta_t`, equivalente al tipo `int`. A questo punto possiamo fare:

```
eta_t eta_mario = 67;
```

Equivalente a

```
int eta_mario = 67;
```

typedef e struct

I **typedef** sono particolarmente utili per creare alias per le **struct** (e per gli **enum**, che vedremo dopo).

```
struct studente_struct {  
    int matricola;  
    int anno_nascita;  
};  
typedef struct studente_struct studente;
```

typedef e struct

I **typedef** sono particolarmente utili per creare alias per le **struct** (e per gli **enum**, che vedremo dopo).

```
struct studente_struct{
    int matricola;
    int anno_nascita;
};
typedef struct studente_struct studente;

int main(){
    studente giorgio;
    giorgio.matricola = 666;
}
```


typedef e struct

È anche possibile definire una struct (anonima) direttamente come **typedef**. Questo è il metodo consigliato.

```
typedef struct {  
    int matricola;  
    int anno_nascita;  
} studente;  
  
int main() {  
    studente giorgio;  
    giorgio.matricola = 666;  
}
```

enum

enum: utile se vogliamo che un **tipo** abbia un range molto limitato e predefinito di valori.

```
enum colore_gatto {  
    rosso,  
    nero,  
    tigrato  
};
```

nel main:

```
enum colore_gatto colore;  
colore = rosso;
```

enum

enum: utile se vogliamo che un **tipo** abbia un range molto limitato e predefinito di valori.

```
enum colore_gatto {           nel main:
    rosso,
    nero,
    tigrato
};                             enum colore_gatto colore;
                                colore = rosso;
```

Come con le **struct**, è possibile definire un **enum** e il suo **typedef** in un solo comando. Questo metodo è consigliato.

```
typedef enum {               nel main:
    rosso,
    nero,
    tigrato
} colore_gatto;             colore_gatto colore;
                                colore = rosso;
```

enum - esempio

Esempio dell'utilizzo di **enum** e **struct**:

```
typedef enum {  
    rosso ,  
    nero ,  
    tigrato  
} colore_gatto ;
```

enum - esempio

Esempio dell'utilizzo di **enum** e **struct**:

```
typedef enum {  
    rosso ,  
    nero ,  
    tigrato  
} colore_gatto ;  
  
typedef struct {  
    colore_gatto colore ;  
    int eta ;  
    double peso ;  
} gatto ;
```

enum - esempio

Esempio dell'utilizzo di **enum** e **struct**:

```
typedef enum {
    rosso ,
    nero ,
    tigrato
} colore_gatto;

typedef struct {
    colore_gatto colore;
    int eta;
    double peso;
} gatto;

int main() {
    gatto felix;
    felix.colore = rosso;
    if (felix.colore != nero)
        printf("Volevo un gatto nero!");
}
```

Outline

- 1 Tipi di dato personalizzati
 - struct
 - enum e typedef
- 2 Array di struct

Array di struct

```
typedef struct {  
    int eta;           // eta' (anni felini)  
    double peso;      // peso (Kg)  
}gatto;  
  
int main(){  
    gatto gatti[10];  
    int i;  
    for(i = 0; i < 10; i++){  
        gatti[i].eta = i + 2;  
    }  
    return 0;  
}
```

Gli array di **struct** funzionano esattamente come gli array di tipi primitivi.