UNIVERSITÀ DI PISA

PROGRAMMAZIONE 1
Corso B

tipi aggregati
manipolazione di bit
file

corrado priami - corrado.priami@unipi.it

oggi vedremo

1. strutture
2. unioni
3. enumerazioni
4. manipolazione di bit
5. campi di bit
6. file

UNIVERSITÀ
DI PISA

PROGRAMMAZIONE 1
Corso B

strutture

corrado priami - corrado.priami@unipi.it

strutture — definizione

una struttura è una collezione di variabili collegate (anche di tipi diversi) con un nome unico (in altri linguaggi è tipicamente chiamata record) — è un **tipo di dato derivato**

definisce un nuovo tipo di dato - non alloca memoria

nome o tag

membri (o campi del record)

```
//  main.c
//  struct
#include <stdio.h>

struct indirizzo {
    char via[20];
    char numero[5];
    unsigned short cap;
    char citta [20];
    char stato [20];
};

struct persona {
    char nome[20];
    char cognome[20];
    unsigned short eta;
    char genere;
    double salario;
    struct indirizzo indirizzo;
    struct persona *CTOPtr;
    // struct persona CTO;
} persona1, persona;

struct {
    struct persona CEO;
    struct persona CTO;
    struct persona CSO;
    struct persona CFO;
} C_level_roles, ruoli[5];

int main(void) {
    struct persona impiegati[100], *personaPtr;
}
```

`struct` autoreferenziale (self-referential)

all'interno di struct posso usare nomi che ho già usato all'esterno per definire nuove variabili

posso inserire puntatori a struct del tipo di quella che sto definendo — servirà per creare strutture dati dinamiche

Field has incomplete type 'struct persona'

dichiarazione di variabili di tipo `struct persona`

il nome è opzionale — se omesso le variabili possono essere definite solo contestualmente alla definizione della `struct`

assegnare una `struct` di un tipo a una `struct` di tipo diverso

definire una `struct` all'interno di un'altra `struct` o dichiarare una variabile di tipo `struct` all'interno dello stesso tipo `struct`

omettere `;` dopo la `}` di chiusura della definizione di `struct`

fornire sempre un nome per le `struct` in modo da poter dichiarare le variabili vicino a dove verranno usate (principio del privilegio minimo)

strutture - inizializzazione

se nella lista avete meno inizializzatori che membri della struct, tutti i membri che seguono l'ultimo inizializzatore sono inizializzati a **0** (oppure **NULL** se sono puntatori)

le variabili struct definite fuori dalle funzioni sono inizializzate automaticamente a 0 o NULL

le variabili struct definite dentro le funzioni non sono inizializzate automaticamente

```c
//  main.c
//  struct
#include <stdio.h>

struct indirizzo {
    char via[20];
    char numero[5];
    char apt[5];
    unsigned int cap;
    char citta[20];
    char stato[20];
};
struct indirizzo abitazione = {"El Camino Real", "4250",
"A305", 94306, "Palo Alto", "California"};

struct persona {
    char nome[20];
    char cognome[20];
    unsigned short eta;
    char genere;
    double salario;
    struct indirizzo indirizzo;
    struct persona *CTOPtr;
    // struct persona CTO;
} persona1, persona;
struct persona persona = {"John"};
struct persona persona1 = {"Bob", "Michaelson", 52, 'M',
0};

struct {
    struct persona CEO;
    struct persona CTO;
    struct persona CSO;
    struct persona CFO;
} C_level_roles, ruoli[5];

int main(void) {
    struct date {
        int day;
        int month;
        int year;
    } inizio, fine;
    struct project {
        char name[20];
        int code;
        struct date start, end;
        double cost;
    } progetto, progetto1;
    persona1.indirizzo = abitazione;
}
```

strutture - accesso . e ->

si accede ai membri di una **struct** mediante il nome della variabile di tipo struct con l'operatore ., oppure mediante un puntatore alla **struct** con l'operatore freccia (arrow).

```
    printf("Persona1\n%s %s %d %c %f\n%s %s %s\n%d %s\n%s\n%p\n",
persona1.nome, persona1.cognome, persona1.eta, persona1.genere,
persona1.salario, persona1.indirizzo.via,
persona1.indirizzo.numero, persona1.indirizzo.apt,
persona1.indirizzo.cap, persona1.indirizzo.citta,
persona1.indirizzo.stato, persona1.CTOPtr);
    printf("C_LEVEL_ROLES - CEO\n%s %s\n", C_level_roles.CEO.nome,
C_level_roles.CEO.cognome);
    printf("progetto\n%s\n", progetto.name);
```

```
Persona1
Bob Michaelson 54 M 0.000000
El Camino Real 4250 A305
94306 Palo Alto
California
0x0
C_LEVEL_ROLES - CEO

progetto
h\366\277\357\376
```

```
struct persona *una_personaPtr = &persona1;
    printf("Persona1 con puntatore\n%s %s %d %c %f\n%s %s %s\n%d
%s\n%s\n%p\n", una_personaPtr->nome, una_personaPtr->cognome,
una_personaPtr->eta, una_personaPtr->genere, una_personaPtr->salario,
una_personaPtr->indirizzo.via, una_personaPtr->indirizzo.numero,
una_personaPtr->indirizzo.apt, una_personaPtr->indirizzo.cap,
una_personaPtr->indirizzo.citta, una_personaPtr->indirizzo.stato,
una_personaPtr->CTOPtr);
```

```
Persona1 con puntatore
Bob Michaelson 54 M 0.000000
El Camino Real 4250 A305
94306 Palo Alto
California
0x0
```

```
//  main.c
//  struct
#include <stdio.h>

struct indirizzo {
    char via[20];
    char numero[5];
    char apt[5];
    unsigned int cap;
    char citta[20];
    char stato[20];
};
struct indirizzo abitazione = {"El Camino Real", "4250",
"A305", 94306, "Palo Alto", "California"};

struct persona {
    char nome[20];
    char cognome[20];
    unsigned short eta;
    char genere;
    double salario;
    struct indirizzo indirizzo;
    struct persona *CTOPtr;
    // struct persona CTO;
} persona1, persona;
struct persona persona = {"John"};
struct persona persona1 = {"Bob", "Michaelson", 52, 'M',
0};

struct {
    struct persona CEO;
    struct persona CTO;
    struct persona CSO;
    struct persona CFO;
} C_level_roles, ruoli[5];

int main(void) {
    struct date {
        int day;
        int month;
        int year;
    } inizio, fine;
    struct project {
        char name[20];
        int code;
        struct date start, end;
        double cost;
    } progetto, progetto1;
    persona1.indirizzo = abitazione;
}
```

operatori accesso strutture

Precedenze	Operatore	arità	associatività
1	**(exp)**	1	annidate: dall'interno all'esterno non annidate: da sinistra a destra
2	**. -> [] () = ++ post - - post**	1	da sinistra a destra
3	**& * + - ! ++ pre - - pre (**cast**)**	1	da destra a sinistra
4	*** / %**	2	da sinistra a destra
5	**+ -**	2	da sinistra a destra
6	**> >= < <=**	2	da sinistra a destra
7	**!= ==**	2	da sinistra a destra
8	**&&**	2	da sinistra a destra
9	**||**	2	da sinistra a destra
10	**?:**	3	da destra a sinistra
11	**= += -= /= *= %=**	2	da destra a sinistra

strutture - operazioni

le **operazioni** possibili sulle `struct` sono:

1. assegnare variabili `struct` ad altre variabili `struct` dello stesso tipo — nel caso di puntatori si copia solo l'indirizzo memorizzato nel puntatore
2. ottenere l'indirizzo di una variabile `struct` con l'operatore `&`
3. accedere ai membri delle variabili `struct`
4. determinarne la dimensione delle variabili `struct` con l'operatore `sizeof`

parola di memoria: è l'unità minima utilizzata per memorizzare informazione in memoria e dipende dalle architetture — tipicamente sono 4 o 8 byte

allineamento di memoria: a seconda delle architetture, i tipi di dato possono essere memorizzati allineati alla parola o alla semi parola

poiché l'allineamento della memoria dipende dalle architetture utilizzate, anche la rappresentazione delle `struct` dipende dalle architetture utilizzate

variabili `struct` non possono essere confrontate con gli operatori == e != perché i loro membri non sono necessariamente memorizzati in locazioni contigue di memoria

assegnare una `struct` di un tipo a una `struct` di tipo diverso

```c
struct pippo {
    char c;
    int x;
};

struct pippo s;

int main(void) {
   s.c = 'b';
   s.x = 11;
}
```

strutture e funzioni

intere strutture possono essere passate alle funzioni oppure membri di strutture oppure puntatori a strutture

strutture o membri di strutture sono passati per valore

passare struct per riferimento è più efficiente perché non richiede di copiare potenzialmente grandi quantità di dati

```
struct progetti_persone {
    int prj[10];
    char name[20];
} prj_John = {{1,1,1,1,1,1,1,1,1,1}, "John"};


void change_array(struct progetti_persone s, size_t dim);

int main(void) {

    change_array(prj_John,10);
    printf("\nDopo change_array\n");
    for (size_t j = 0; j < 10; j++) {
        printf("a[%lu] = %d ", j, prj_John.prj[j]);
    }
}

void change_array(struct progetti_persone s, size_t dim) {
    printf("Dentro change_array\n");
    for (size_t j = 0; j < dim; j++) {
        s.prj[j]++;
        printf("a[%lu] = %d ", j, s.prj[j]);
    }
}
```

se passo una struttura per valore che contiene l'array, anche l'array è passato per valore

array di strutture sono invece passati per riferimento come ogni array

```
Dentro change_array
a[0] = 2 a[1] = 2 a[2] = 2 a[3] = 2 a[4] = 2 a[5] = 2 a[6] = 2 a[7] = 2 a[8] = 2 a[9] = 2
Dopo change_array
a[0] = 1 a[1] = 1 a[2] = 1 a[3] = 1 a[4] = 1 a[5] = 1 a[6] = 1 a[7] = 1 a[8] = 1 a[9] = 1
```

typedef

consente di creare sinonimi (alias) per tipi precedentemente definiti

```c
struct indirizzo {
    char via[20];
    char numero[5];
    char apt[5];
    unsigned int cap;
    char citta[20];
    char stato[20];
};

typedef struct indirizzo Address;

Address a;
```

da qui in poi posso usare **address** oppure **struct indirizzo** indifferentemente

la variabile a ha tipo **struct indirizzo**, ma evito di dover scrivere ogni volta che dichiaro una variabile **struct**

```c
typedef struct {
    int a;
    char b;
} Esempio;

Esempio struttura = {1, 'a'};
```

in questo modo uso **typedef** per dare un nome ad una struttura senza nome e poi lo uso per definire le variabili

utilizzare sempre la prima lettera maiuscola per gli alias dei tipi definiti con **typedef** per distinguerli facilmente dai tipi primitivi

UNIVERSITÀ
DI PISA

PROGRAMMAZIONE 1
Corso B

unioni

corrado priami - corrado.priami@unipi.it

union

le unioni sono tipi derivati come le strutture e possono contenere membri di tipi non omogenei, ma tutti i membri condividono un numero di byte comuni grande almeno quanto il tipo di dato più grande

in ogni istante solo un membro è attivo e in fase di inizializzazione si può inizializzare solo il primo membro

le operazioni sono assegnare una unione ad un'altra se sono dello stesso tipo, ottenere l'indirizzo di una variabile unione con l'operatore `&` e accedere ai membri dell'unione con gli operatori `.` e `->`

la memoria necessaria è dipendente dall'architettura, compilatore e OS per l'allineamento alla parola scelto

riferire un membro quando non è attivo

le unioni possono essere passate come parametri di funzioni e sono gestite per valore come per le strutture

```c
/  main.c
//  union
#include <stdio.h>
#include <limits.h>

union integral_types {
    int x;
    double y;
    char a;
};

int main(void) {
    union integral_types itu = {INT_MAX};
    printf("I membri dell\'unione sono:\n x = %d, y = %f, a = %c\n", itu.x, itu.y, itu.a);
    itu.y = 12.21;
    printf("I membri dell\'unione sono:\n x = %d, y = %f, a = %c\n", itu.x, itu.y, itu.a);
}
```

```
I membri dell'unione sono:
 x = 2147483647, y = 0.000000, a = \377
I membri dell'unione sono:
 x = 515396076, y = 12.210000, a = \354
```

UNIVERSITÀ DI PISA

PROGRAMMAZIONE 1
Corso B

enumerazioni costanti

corrado priami - corrado.priami@unipi.it

enum

le enumerazioni costanti sono insiemi di interi rappresentati da identificatori che di default partono assegnato **0** al primo identificatore e poi incremento di uno per ogni identificatore successivo

utilizzare solo lettere maiuscole per le costanti dell'enumerazione per distinguerle subito dentro il programma

assegnare un valore ad una costante dell'enumerazione dopo l'inizializzazione

```c
//  main.c
//  union
#include <stdio.h>

union integral_types {
    int x;
    double y;
    char a;
};

enum giorni {
    LUN=5, MAR, MER, GIO, VEN = 22, SAB, DOM
};

int main(void) {
    printf("%d %d %d %d %d %d %d\n", LUN, MAR, MER, GIO, VEN, SAB, DOM);
    for (enum giorni day = LUN; day <= DOM; day++) {
        printf("%d\n", day);
    }
}
```

```
5 6 7 8 22 23 24
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
```

UNIVERSITÀ
DI PISA

PROGRAMMAZIONE 1
Corso B

manipolazione di bit

corrado priami - corrado.priami@unipi.it

operatori sui bit

gli operatori sui bit manipolano la rappresentazione binaria dei tipi di dato numerici e carattere

operatore	Descrizione
&	AND — confronta due operandi bit a bit e restituisce AND dei singoli bit
|	OR — confronta due operandi bit a bit e restituisce OR dei singoli bit
^	XOR — confronta due operandi bit a bit e restituisce XOR dei singoli bit (vero solo se i bit sono diversi)
<<	sposta i bit del primo operando a sinistra del numero di posizioni indicate dal secondo operando e riempe le posizioni lasciate vuote con 0
>>	sposta i bit del primo operando a destra del numero di posizioni indicate dal secondo operando e riempe le posizioni lasciate vuote con 0
~	complementa (**1** in **0** e **0** in **1**) tutti i bit del singolo operando
&=	`x &= y` è equivalente a `x = x & y`
|=	`x |= y` è equivalente a `x = x | y`
^=	`x ^= y` è equivalente a `x = x ^ y`
<<=	`x <<= y` è equivalente a `x = x << y`
>>=	`x >>= y` è equivalente a `x = x >> y`

gli operatori sui bit dipendono dalla rappresentazione binaria dei dati adottata e quindi sonno dipendenti dal tipo di architettura utilizzata — usare `sizeof` e la costante `CHAR_BIT` per determinare la dimensione in bit dei vari tipi per aumentare portabilità e non inserire valori costanti direttamente

```
sizeof(int)*CHAR_BIT
```

numero di bit in un byte

utilizzare & e | come se fossero AND logico (&&) e OR logico (||)

il risultato di << e >> è indefinito se l'operatore di destra è negativo o più grande del numero di bit disponibili nella rappresentazione

operatori sui bit

Precedenze	Operatore	arità	associatività
1	**(exp)**	1	annidate: dall'interno all'esterno non annidate: da sinistra a destra
2	**. -> [] () = ++ post - - post**	1	da sinistra a destra
3	**& * + - ! ++ pre - - pre ~** (cast)	1	da destra a sinistra
4	*** / %**	2	da sinistra a destra
5	**+ -**	2	da sinistra a destra
6	**>> <<**	2	da sinistra a destra
7	**> >= < <=**	2	da sinistra a destra
8	**!= ==**	2	da sinistra a destra
9	**&**		da sinistra a destra
10	**^**		da sinistra a destra
11	**|**		da sinistra a destra
12	**&&**	2	da sinistra a destra
13	**||**	2	da sinistra a destra
14	**?:**	3	da destra a sinistra
15	**= += -= /= *= %= &= |= ^= <<= >>=**	2	da destra a sinistra

stampa bit

```c
//  main.c
//  bit manipulation
#include <stdio.h>
#include <limits.h>

const int BITS = sizeof(unsigned int)*CHAR_BIT;

void stampa_bit(unsigned int value);

int main(void) {
    unsigned int value = 123456789;
    printf("%9d = ", value);
    stampa_bit(value);
}

void stampa_bit(unsigned int value) {
    unsigned int maschera = 1 << (BITS - 1);
    for (short i = 1; i <= BITS; i++) {
        putchar(value & maschera ? '1' : '0');
        value <<= 1;
        if (i % 8 == 0) {
            putchar(' ');
        }
    }
    putchar('\n');
}
```

```
 123456789 = 00000111 01011011 11001101 00010101
  maschera = 10000000 00000000 00000000 00000000

00001110 10110111 10011010 00101010
10000000 00000000 00000000 00000000
00011101 01101111 00110100 01010100
10000000 00000000 00000000 00000000
00111010 11011110 01101000 10101000
10000000 00000000 00000000 00000000
01110101 10111100 11010001 01010000
10000000 00000000 00000000 00000000
11101011 01111001 10100010 10100000
10000000 00000000 00000000 00000000
11010110 11110011 01000101 01000000
10000000 00000000 00000000 00000000
10101101 11100110 10001010 10000000
10000000 00000000 00000000 00000000
:
:
```

un esempio

```c
//  main.c
//  bit manipulation
#include <stdio.h>
#include <limits.h>

const int BITS = sizeof(unsigned int)*CHAR_BIT;

void stampa_bit(unsigned int value);

int main(void) {
    unsigned int value = 123456789;
    unsigned int value1 = 987654321;
    printf("%9d = ", value);
    stampa_bit(value);
    printf("%9d = ", value1);
    stampa_bit(value1);
    printf("%9s = ", "AND");
    stampa_bit(value & value1);
    printf("%9s = ", "OR");
    stampa_bit(value | value1);
    printf("%9s = ", "XOR");
    stampa_bit(value ^ value1);
    printf("%9s = ", "sin 4");
    stampa_bit(value << 4);
    printf("%9s = ", "dx 4");
    stampa_bit(value >> 4);
    printf("%9s = ", "compl");
    stampa_bit(~value);
}

void stampa_bit(unsigned int value) {
    unsigned int maschera = 1 << (BITS - 1);
    for (short i = 1; i <= BITS; i++) {
        putchar(value & maschera ? '1' : '0');
        value <<= 1;
        if (i % 8 == 0) {
            putchar(' ');
        }
    }
    putchar('\n');
}
```

```
123456789 = 00000111 01011011 11001101 00010101
987654321 = 00111010 11011110 01101000 10110001
      AND = 00000010 01011010 01001000 00010001
       OR = 00111111 11011111 11101101 10110101
      XOR = 00111101 10000101 10100101 10100100
    sin 4 = 01110101 10111100 11010001 01010000
     dx 4 = 00000000 01110101 10111100 11010001
    compl = 11111000 10100100 00110010 11101010
```

campi di bit

è possibile definire nelle strutture la dimensione in bit dei membri se sono di tipo intero o carattere — la dimensione deve essere <= dei bit usati per rappresentare il tipo di dato

```
struct bit_dei_campi {
    int x : 10;
    char y : 2;
};
```

possono essere usati per codifiche (ad esempio `y` può codificare `4` alternative — es colori — usando solo `2` bit

```
struct bit_dei_campi {
    char y : 4;
    int : 0;
    int x : 10;
  };
```

campi senza nome su `0` bit sono usati per forzare la rappresentazione allineata alla parola — `x` viene allineato alla parola successiva di `y`

i campi di bit riducono l'occupazione di memoria, ma possono generare codice più lento perché il compilatore deve aggiungere istruzioni aggiuntivi per manipolare dati di dimensioni nn standard

tentare di accedere ai bit di un campo di bit come se fosse un array (mediante indici) oppure cercare di ottenere l'indirizzo dei campi di bit con & (non definito)

la rappresentazione dei campi di bit dipende dall'architettura

UNIVERSITÀ DI PISA

PROGRAMMAZIONE 1
Corso B

file

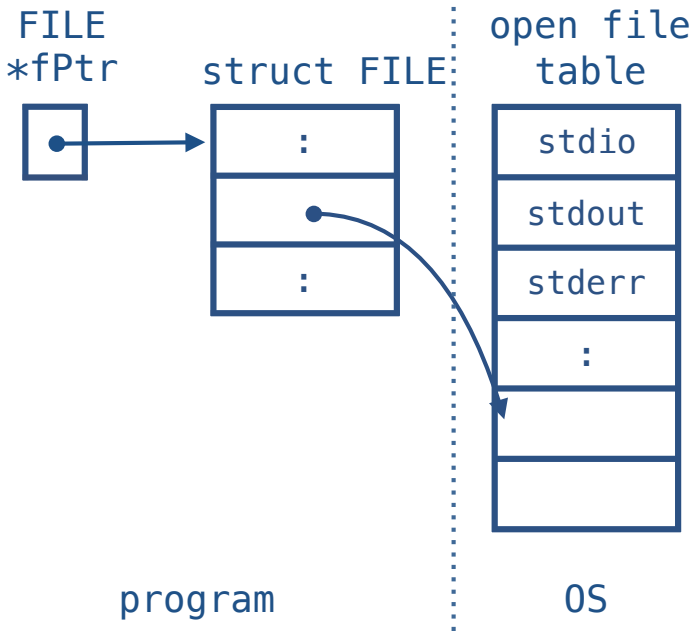
corrado priami - corrado.priami@unipi.it

introduzione

in C un **file** è uno **stream** (canale tra una sorgente e una destinazione per comunicare dati — flusso di dati) di byte che termina ad una posizione predefinita dal sistema o con il carattere speciale `EOF` (end of file) — `stdio` è uno stream che permette al programma di comunicare con la tastiera, `stdout` è uno stream che permette al programma di comunicare con il terminale, `stderr` è uno stream che permette di stampare i messaggi di errore — questi 3 sono aperti automaticamente all'avvio del programma

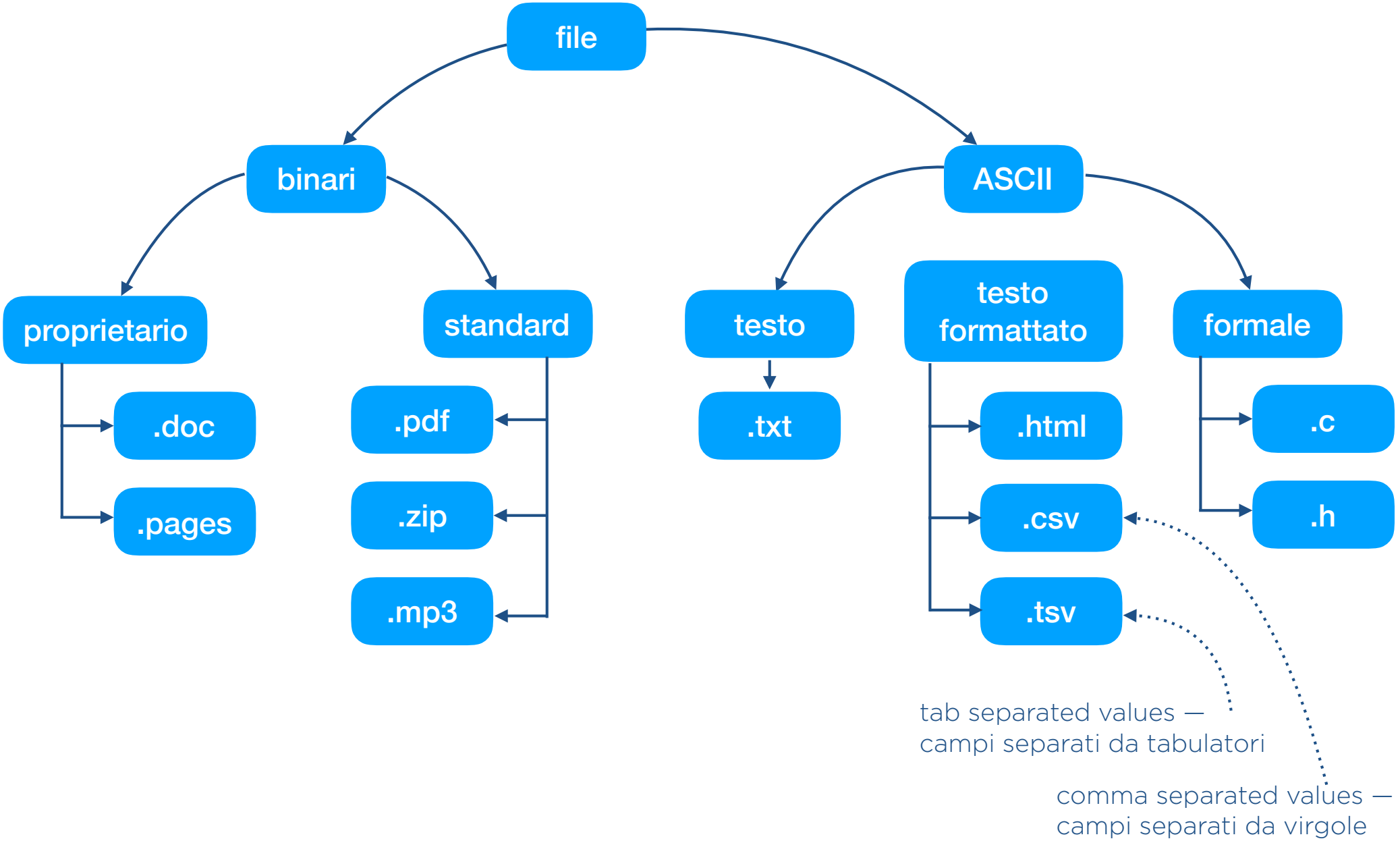
si accede a un file mediante un puntatore a una `struct FILE` definita in `stdio.h`

è un indice a una tabella gestita dal sistema operativo **FCB** (file control block), usata per gestire tutte le operazioni sul file e mantenere la **open file table**




```c
typedefstruct __sFILE {
    unsigned char *_p;   /* current position in (some) buffer */
    int _r;    /* read space left for getc() */
    int _w;    /* write space left for putc() */
    short  _flags;    /* flags, below; this FILE is free if 0 */
    short  _file;     /* fileno, if Unix descriptor, else -1 */
    struct __sbuf _bf;   /* the buffer (at least 1 byte, if !NULL) */
    int _lbfsize;  /* 0 or -_bf._size, for inline putc */

    /* operations */
    void   *_cookie;  /* cookie passed to io functions */
    int (*_close)(void *);
    int (*_read) (void *, char *, int);
    fpos_t (*_seek) (void *, fpos_t, int);
    int (*_write)(void *, const char *, int);

    /* separate buffer for long sequences of ungetc() */
    struct __sbuf _ub;   /* ungetc buffer */
    struct __sFILEX *_extra; /* additions to FILE to not break ABI */
    int _ur;       /* saved _r when _r is counting ungetc data */

    /* tricks to meet minimum requirements even when malloc() fails */
    unsigned char _ubuf[3];  /* guarantee an ungetc() buffer */
    unsigned char _nbuf[1];  /* guarantee a getc() buffer */

    /* separate buffer for fgetln() when line crosses buffer boundary */
    struct __sbuf _lb;   /* buffer for fgetln() */

    /* Unix stdio files get aligned to block boundaries on fseek() */
    int _blksize;  /* stat.st_blksize (may be != _bf._size) */
    fpos_t _offset;   /* current lseek offset (see WARNING) */
} FILE;
```

tipi di file


file
binari
ASCII
proprietario
standard
testo
testo formattato
formale
.doc
.pages
.pdf
.zip
.mp3
.txt
.html
.csv
.tsv
.c
.h
tab separated values —
campi separati da tabulatori
comma separated values —
campi separati da virgole

operazioni

modo	descrizione	modo	descrizione
`r`	apre un file esistente in lettura	`rb`	come `r`, ma file binario
`w`	crea un file in scrittura — se già esiste ne cancella il contenuto e lo apre in scrittura	`wb`	come `w`, ma file binario
`a`	apre un file per appenderne (scrittura) contenuto alla fine	`ab`	come `a`, ma file binario
`r+`	apre un file esistente per modifica (lettura e scrittura)	`rb+`	come `r+`, ma file binario
`w+`	crea un file in lettura e scrittura — se già esiste ne cancella il contenuto e lo apre in scrittura	`wb+`	come `w+`, ma file binario
`a+`	apre un file per modifica (lettura e scrittura), ma scritture sono fatte solo alla fine del file	`ab+`	come `a+`, ma file binario

aprire in lettura un file non esistente, aprire un file in lettura/scrittura senza avere i diritti di accesso, aprire un file in scrittura quando non c'è più spazio sul disco, accedere a un file prima di aprirlo

aprire i file nella giusta modalità per garantire il principio del privilegio minimo

file accesso sequenziale

```
citta, stato
Roma Italia
citta, stato
---------
citta, stato sono Roma e Italia
---------
citta, stato sono Roma e Italia
citta, stato sono San_Mateo e California
```

```c
//  main.c
//  file
#include <stdio.h>

int main(void) {
    FILE *fPtr, *f1Ptr;
    char citta[15];
    char stato[15];
    if ((fPtr = fopen("address.txt", "w")) == NULL) {
        puts("Impossibile aprire file");
    }
    else {
        puts("citta, stato");
        scanf("%15s%15s", citta, stato);
        while (!feof(stdin)) {
            fprintf(fPtr, "%15s %15s", citta, stato);
            puts("citta, stato");
            scanf("%15s%15s", citta, stato);
        }
    }
    fPtr = freopen("address.txt", "r", fPtr);
    puts("---------");
    while (!feof(fPtr)) {
        fscanf(fPtr, "%15s %15s", citta, stato);
        printf("citta, stato sono %s e %s\n", citta, stato);
    }
    puts("---------");
    fPtr = freopen("address.txt", "a+", fPtr);
    fprintf(fPtr, "%15s %15s", "San_Mateo", "California");
    rewind(fPtr);
    while (!feof(fPtr)) {
        fscanf(fPtr, "%15s %15s", citta, stato);
        printf("citta, stato sono %s e %s\n", citta, stato);
    }
    fclose(fPtr);
}
```

definisco 2 puntatori a file

apro il file in scrittura

stdin, **stdout** e **stderr** sono puntatori a file

scrivo sul file

riapro il file in lettura

controllo che il file non sia terminato

leggo dal file

riapro il file in aggiornamento — append alla fine

scrivo sul file

riposiziono il punitore all'inizio del file

chiudo il file

altre funzioni

`fgetc(FILE *)` legge un carattere da un file

`fgetc(stdin)` è equivalente a `getchar()`

`fputc(char, FILE *)` legge un carattere da un file

`fputc('a', stdin)` è equivalente a `putchar('a')`

file accesso casuale

i file ad accesso casuale sono organizzati in slot di dimensione normalmente fissata e consentono di accedervi senza dover scorrere tutto il file — si può calcolare la posizione di un record come offset dall'inizio del file e di una chiave di accesso al record

è possibile inserire nuovi dati senza distruggere quelli già presenti
è possibile leggere i dati sia sequenzialmente che casualmente
è possibile aggiornare i dati
è possibile cancellare record

è possibile posizionare il puntatore del file ad una posizione specifica con `fseek`

al posto `fprintf` e `fscanf` si usano `fwrite` e `fread`

scrivono e leggono il numero di caratteri necessario a rappresentare testualmente il dato (es. `1` si scrive un carattere)

scrivono e leggono un numero di byte passato come argomento

possono scrivere alcuni elementi simultaneamente sul file — per questo hanno un puntatore come parametro

un esempio

la prossima volta

Scrivere un programma C che legge da un file informazioni relativi ad arrivi e partenze dei treni dalle stazioni Toscane. For each line, the file contains the following information (each field contains maximum 20 characters and no spaces):
<departure_station> <departing_time> <destination_station> <arriving_time>

Data una città che può essere una stazione di partenza o di arrivo, il programma calcola quanti treni partono da quella città, quanti ne arrivano e quale è la fascia oraria in cui c'è il maggior traffico nella stazione (somma dei treni che partono ed arrivano). Stampa quindi una tabella di tre colonne in cui nella prima è indicata un'ora, nella seconda quanti treni partono dalla stazione e nella terza quanti ne arrivano. L'ultima riga contiene il totale della seconda e terza colonna. Le colonne devono essere allineate a destra e devono essere separate da 4 spazi per favorire la lettura. L'intestazione delle colonne deve essere ORA, PARTENZE e ARRIVI. Nella colonna ORA l'ultima riga deve contenere TOT.

Un esempio di output potrebbe essere:
Le partenze da Pisa 23 e gli arrivi sono 18

ORA	PARTENZE	ARRIVI
…		
09:00	2	4
10:00	1	1
…		
TOT	23	18

oggi abbiamo visto ...

1. strutture
2. unioni
3. enumerazioni
4. manipolazione di bit
5. campi di bit
6. file

UNIVERSITÀ
DI PISA

PROGRAMMAZIONE 1
Corso B

buona giornata!!!!

corrado priami - corrado.priami@unipi.it