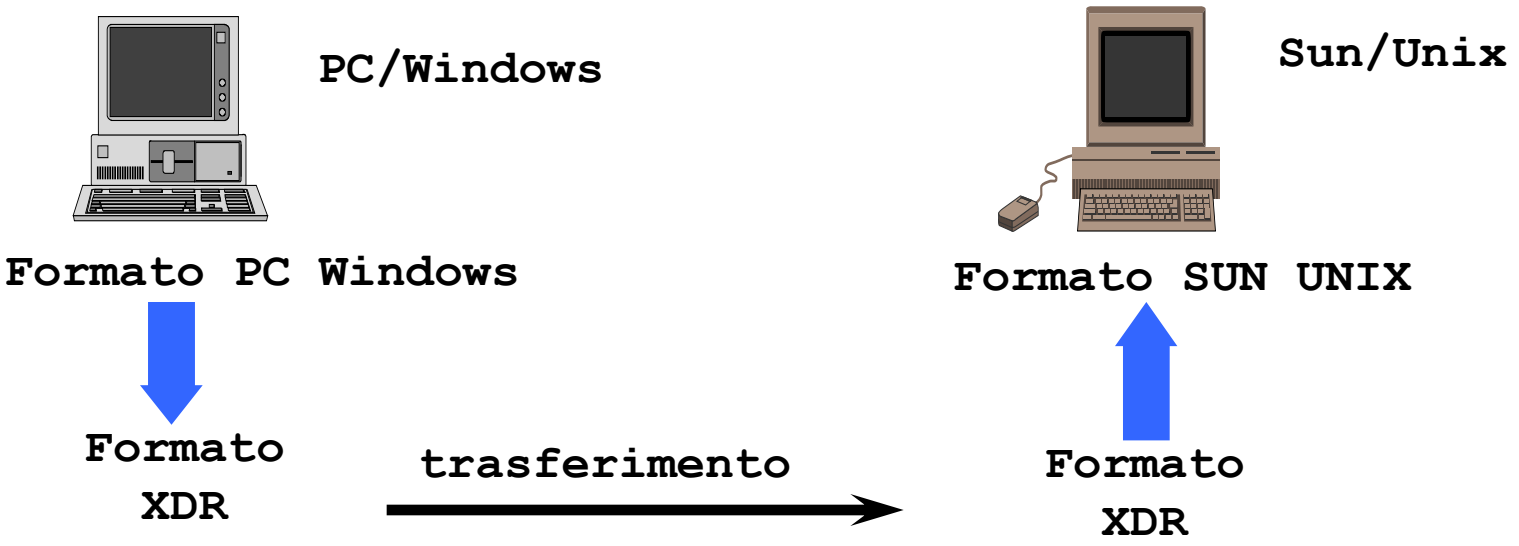
XDR (eXternal Data Representation)

- È uno standard per la descrizione e la codifica dei dati (rappresentazione aperta per lo scambio di dati tra piattaforme eterogenee)
- Originariamente introdotto dalla SUN
- Standardizzato come RFC 1014, RFC 1832, RFC 4506 (nel 2006)
- Svolge un ruolo simile a quello di altri linguaggi per la descrizione astratta dei dati (ASN.1)

Tipico uso di XDR

- XDR è utile come linguaggio comune per trasferire dati tra macchine con architetture e/o sistemi operativi diversi



Concetti basilari

- XDR assume che il byte sia l'unità **portabile** (non ci siano ambiguità sull'interpretazione dei bit)
- La parola di 4 byte è l'unità fondamentale di tutte le rappresentazioni XDR (compromesso per minimizzare gli allineamenti e la dimensione dei messaggi)
- Lo standard XDR definisce:
  - **Un linguaggio C-like per descrivere i tipi di dato (solo i dati!)**
  - **Una rappresentazione univoca per ciascun dato descritto** (sequenza ordinata di byte, dove il primo byte - o byte 0 - è il primo ad essere trasferito/memorizzato)

Tipi scalari riconosciuti

`int`	32 bit	interi con segno
`unsigned int`	32 bit	interi senza segno
`bool`	32 bit	booleani (0 falso, 1 vero)
`enum`	arbitr.	tipi dati per enumerazione
`hyper`	64 bit	interi lunghi con segno
`unsigned hyper`	64 bit	interi lunghi senza segno
`float`	32 bit	numeri in virgola mobile
`double`	64 bit	numeri in virgola mobile

Tipi strutturati riconosciuti

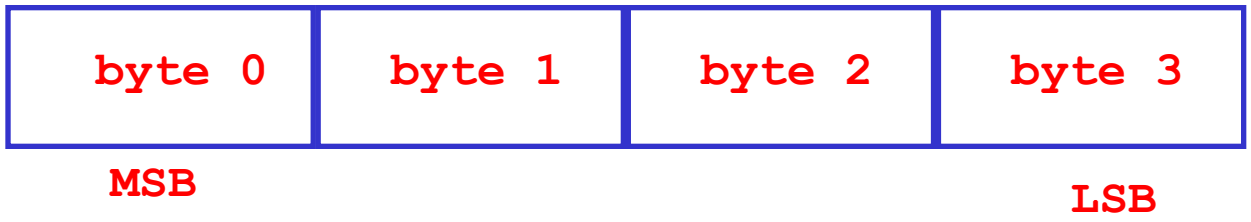
`fixed array`	arb.	vettore con un numero fisso di elementi
`counted array`	arb.	vettore con numero di elementi variabile
`string`	arb.	stringhe ASCII
`opaque`	arb.	vettore di dati non convertiti
`struct`	arb.	struttura (come in C)
`union`	arb.	struttura con formati variabili (come union in C)
`void`	0 bit	usato quando non è presente un dato opzionale
`const`	arb.	costante simbolica
`Optional data`	arb.	dato opzionale

Interi con segno

- Sintassi:

  int *identifier*

- Rappresentazione: (big endian)



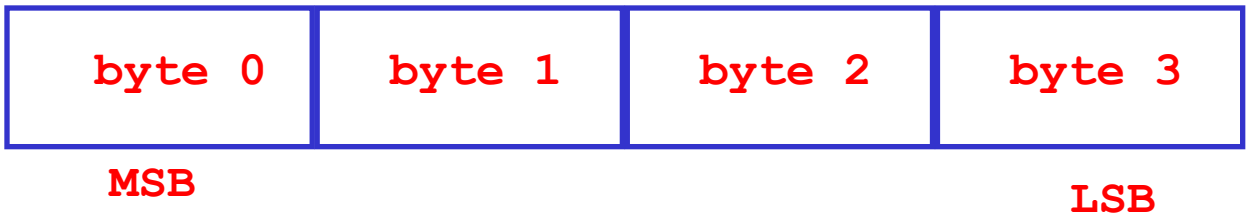
- Codifica: Complemento a 2

Interi senza segno

- Sintassi:

  unsigned int *identifier*

- Rappresentazione (big endian):



- Codifica: Binario puro

Dati definiti per enumerazione

- Sintassi:

  enum { *name_id* = *const* , ... }
  identifier

- Rappresentazione e Codifica: quella degli interi con segno (sono legali però solo i valori definiti)
- Esempio:

```
enum {BIANCO=0, NERO=1, ROSSO=2,
  BLU=3} colore
```

Dati booleani

- Sintassi:

  bool *identifier*

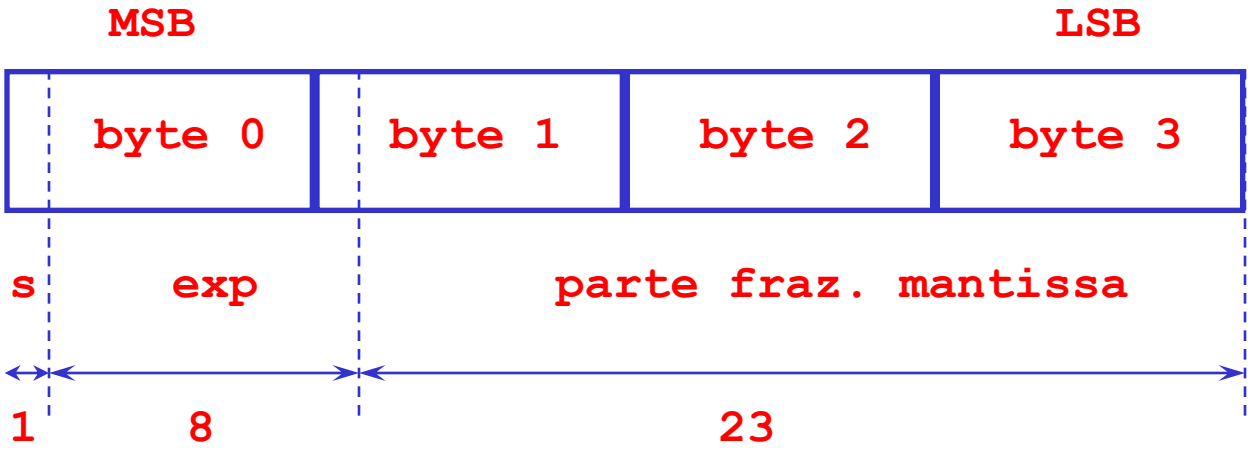
- Equivale a:
  - **`enum {FALSE=0, TRUE=1 }`** ***`identifier`***

Numeri floating point

- Sintassi:

  float *identifier*

- Rappresentazione e Codifica: IEEE 754 singola precisione



Rappresentazioni estese

- Interi su 8 byte (complemento a 2)

  hyper *identifier*

  unsigned hyper *identifier*

- Floating point doppia precisione (IEEE 754 d.p. - 8 byte): 1 segno, 11 esponente, 52 mantissa:

  double *identifier*

- Floating point quadrupla precisione (16 byte): 1 segno, 15 esponente, 112 mantissa:

  quadruple *identifier*

Void

- È l'elemento nullo
- Sintassi:

  void

- Rappresentazione e Codifica:

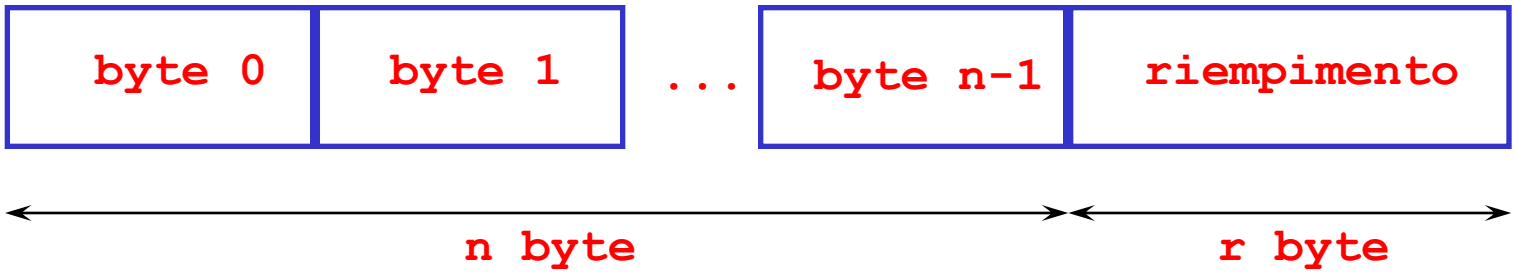
Il void è rappresentato su 0 byte.

Dati "opachi" a lunghezza fissa

- Sono sequenze di byte di cui non sono rilevanti il significato e la codifica.
- Sintassi:

  opaque *identifier* **[***n***]**

- Rappresentazione:



`r varia da 0 a 3  ((n+r) mod 4 = 0)`

Dati "opachi" a lunghezza variabile

- Sintassi:

  opaque *identifier* **<** *n* **>** (n è la lunghezza max)

  opaque *identifier* **< >** (lungh. max = 2^{32}-1)

- Rappresentazione:

La lunghezza effettiva L è rappresentata come unsigned int

Il resto è rappresentato come la sequenza a lungh. fissa

Stringhe

- Sono rappresentate come i dati opachi a lunghezza variabile
- Sintassi:

  string *identifier* **<** *n* **>** (n è la lunghezza max)

  string *identifier* **< >** (lungh. max = 2^{32}-1)
- Rappresentazione: Il byte 4 (il primo dopo la lunghezza) è il primo carattere della stringa.
- Codifica: ASCII su 1 byte

Array a dimensione fissa

- Sono sequenze di dati di tipo omogeneo (ciascuno rappresentato su un multiplo di 4 byte)
- Sintassi:

  tipo *identifier* [*n*]

- Rappresentazione:

  Semplice sequenza delle singole rappresentazioni (gli elementi possono avere dimensioni diverse!)

Array a dimensione variabile

- Sintassi:

  tipo identifier **<** *n* **>** (lungh. max=n)

  tipo identifier **< >** (lungh. max = 2^{32}-1)

- Rappresentazione:

La lunghezza effettiva L è rappresentata come unsigned int

Il resto è rappresentato come l'array a dimensione fissa

Strutture

- Sono sequenze di dati di tipo diverso (ciascuno rappresentato su un multiplo di 4 byte)
- Sintassi:

  struct { *elemento_1* **;**

  ...

  elemento_n **; }** *identifier*

- Rappresentazione:

  Semplice sequenza delle singole rappresentazioni

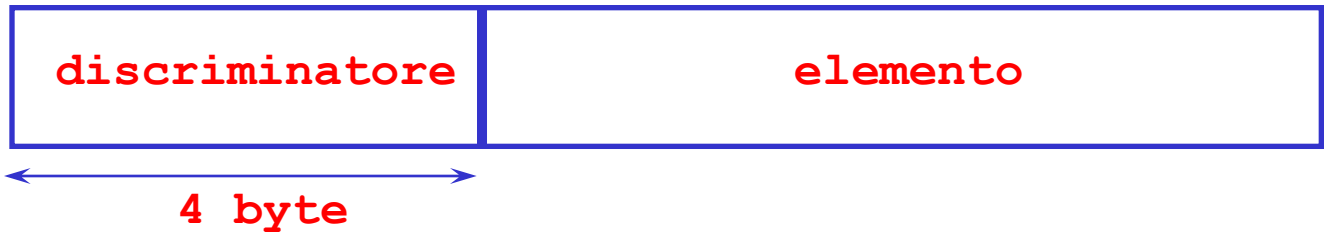
Union

- Sono costituite da un elemento discriminatore, seguito da un elemento il cui tipo dipende dal valore del discriminatore
- Sintassi:

union switch(*discriminatore*) {
case *valore_1* : *elemento_1* ;
...
case *valore_n* : *elemento_n* ;
default : *elemento_def* ; } *identifier*

Union (cont)

- Il discriminatore è di tipo int o unsigned int o enum
- Il ramo default è opzionale
- Rappresentazione:



l'elemento rappresentato dopo il discriminatore è quello del ramo case relativo al valore del discriminatore.

Costanti e Typedef

- È possibile assegnare nomi simbolici a valori e a tipi di dato:
- Sintassi per le costanti:

  const *identifier* = *value*
- Sintassi per i tipi (come in linguaggio C):

  typedef *declaration*

  oppure, per struct, union o enum, si può usare la forma:

  enum bool { FALSE = 0; TRUE = 1; }

Elemento opzionale

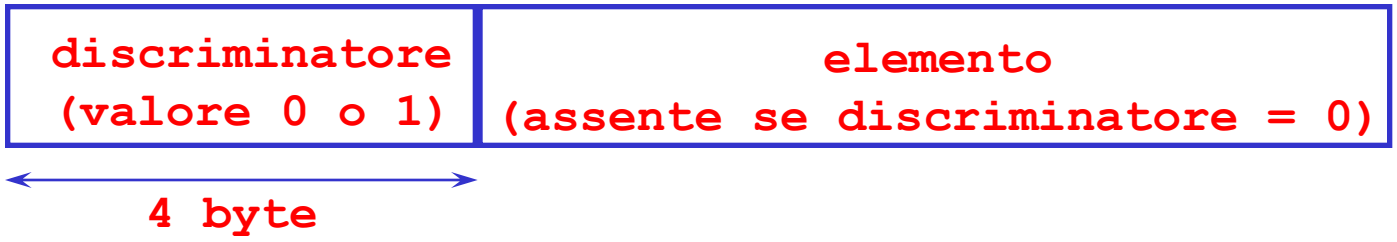
- Sintassi:

  *type * identifier*

- Equivale ad un array di dimensione variabile, la cui dimensione può essere solo 0 oppure 1:

  type identifier <1>

- Rappresentazione:

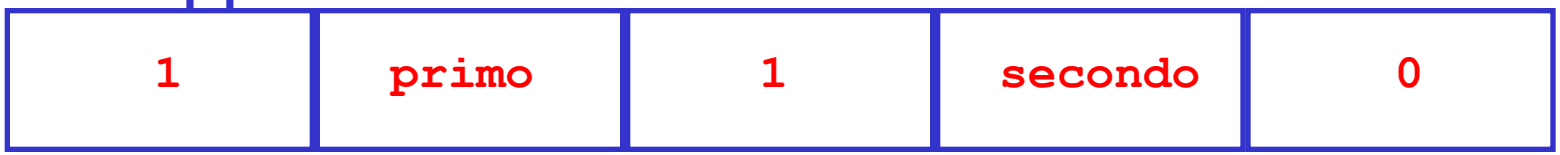


Liste

- Vengono definite recursivamente usando i typedef e gli elementi opzionali.
- Esempio: lista di interi

```
struct * intlist {
        int             item;
        intlist         next;
}
```

- Rappresentazione di una lista di 2 interi:



Esempio: un semplice tipo XDR per rappresentare un file (1)

```
/* Definitions of constants:
*/
const MAXUSERNAME = 32;    /* max length of a user name*/
const MAXFILELEN = 65535; /* max length of a file     */
const MAXNAMELEN = 255;    /* max length of a file name*/

/* Types of files:
*/
enum filekind {
   TEXT = 0,         /* ascii data */
   DATA = 1,         /* raw data   */
   EXEC = 2          /* executable */
};
```

Esempio: un semplice tipo XDR per rappresentare un file (2)

```
/* File information, per kind of file:
*/
union filetype switch (filekind kind) {
    case TEXT: void;       /* no extra information */
    case DATA:
         string creator<MAXNAMELEN>; /* data creator  */
    case EXEC:
         string interpreter<MAXNAMELEN>; /* data interpreter  */
};

/* A complete file:
*/
struct file {
     string filename<MAXNAMELEN>; /* name of file    */
     filetype type;               /* info about file */
     string owner<MAXUSERNAME>;   /* owner of file   */
     opaque data<MAXFILELEN>;     /* file data       */
};
```

Libreria XDR

- La manipolazione dei dati in formato XDR viene realizzata da un'apposita libreria (tipicamente disponibile in ambiente UNIX, vedi guida **man**)
- Tipiche operazioni:
  - conversioni da/verso tipi *C*
- La libreria normalmente usa il *buffer paradigm*:
  - il dato XDR viene costruito/memorizzato in un buffer (lungo in modo tale da contenere i dati)
- Esistono strumenti per generare automaticamente il codice di codifica e decodifica di tipi XDR complessi

Tipica Codifica XDR

```
#include <rpc/xdr.h>
#define BUFSIZE 4000
XDR xdrs;   /* pointer to XDR stream */
char buf[BUFSIZE];/* buffer for XDR data */

/* create stream */
xdrmem_create(&xdrs, buf, BUFSIZE, XDR_ENCODE);

/* append data to XDR stream */
int i=800,j; float a=3.14;
char *p="ciao mondo";
j=strlen(p);
xdr_int(&xdrs, &i); /* convert/append integer i */
xdr_float(&xdrs,&a);
xdr_bytes(&xdrs,&p,&j,80); /* convert/append string, max
   len=80 */
int len = xdr_getpos(&xdrs); /* the amount of buf used*/
xdr_destroy(&xdrs); /* destroy (free) stream */
```

Tipica Decodifica XDR

```
#include <rpc/xdr.h>
#define BUFSIZE 4000
XDR xdrs;   /* pointer to XDR stream */
char buf[BUFSIZE];/* buffer for XDR data */

/* create stream */
xdrmem_create(xdrs, buf, BUFSIZE, XDR_DECODE);

/* extract data from XDR stream */
int i, k; float b; char *q;

xdr_int(&xdrs, &i); /*extract integer i */
xdr_float(&xdrs,&b);
xdr_bytes(&xdrs,&q,&k,80);

/* destroy (free) stream */
xdr_destroy(&xdrs);
```

Come avviene l'invio/ricezione (1)

Esistono due possibilità

1. Una volta riempito un buffer XDR lo si spedisce sul socket (datagram o stream)
   - Il ricevitore legge il buffer dal socket e decodifica con XDR il contenuto

Come avviene l'invio/ricezione (2)

2. Con TCP, si può leggere/scrivere direttamente quando si codifica/decodifica con XDR, bisogna:
   - Usare **`fdopen`** per associare uno stream STDIO al socket
   - Usare **`xdrstdio_create`** per XDR
   - Quando si converte, il risultato viene **automaticamente** spedito sul socket
   - Quando si decodifica, il sistema aspetta di estrarre dal socket i dati necessari (così non è necessario conoscere le dimensioni dei dati in XDR)
   - Se si vuole accedere direttamente al socket si DEVONO usare le funzioni per STDIO (**`fread, fwrite`**) e non le send/recv ecc. perché l'input viene bufferizzato
   - In scrittura, ogni volta che il messaggio è stato preparato, FORZARE la scrittura sul socket con **`fflush`** se è stata usata la **`xdrstdio_create`**

Generazione del codice

- Problema: come scrivere codice portabile (che non dipende dalla dimensione dei tipi in C)
- Una possibile soluzione: generazione del codice
  - Scrivere in XDR language i dati di interesse
  - Compilare l'XDR (tramite es. **`rpcgen`**) per generare:
    - header file (definizioni con tipi C)
    - C file (funzioni di codifica e decodifica)
  - Scrivere il resto del programma includendo gli header generati e usando le funzioni di codifica e decodifica generate

Esempio di generazione e uso codice

- XDR (types.x):

```
enum operation { ENC = 0, DEC = 1 };
struct Request { operation op;  float data<>; };
```

- Comandi:

```
rpcgen -o types.c -c  types.x    # generate types.c
rpcgen -o types.h -h  types.x    # generate types.h
```

- Uso:

```
#include "types.h"
int main(...) {...
 XDR xdrs; float *v = malloc(...); Request req;
 xdrmem_create(&xdrs, buf, BUFSIZE, XDR_ENCODE);
 // Fill up structure:
 req.op = ENC; req.data_len = mylen; req.data_val = v;
 if (!xdr_Request(&xdrs, &req)) { printf("Error"); }
 send(sock, buf, xdr_getpos(&xdrs), 0);
 ... }
```