0author]

Fondamenti di informatica II
per i corsi di diploma universitario in Ingegneria Biomedica e Ingegneria Elettronica anno accademico 1998/99

Agenda delle lezioni dell'A.A. 1998-99

Locazione originaria: <URL:http://fly.cnuce.cnr.it/didattica/agenda98.html>.

Abbreviazioni usate nell'agenda:

(2L)
: 2 ore di lezione (sviluppo di nuovi argomenti)

(1E)
: 1 ora di esercitazioni (esercizi in classe)

(1S)
: 1 ora di laboratorio (pratica al calcolatore)

D: 9.1, 9.5…; C:7.6
: Argomenti trattati nei testi di riferimento Domenici, capitoli 9.1 e tutto il 9.5, e a Ceri, capitolo 7.6.

Negli esempi di codice riportati si usa la notazione raccomandata dello standard del C++ per gli include standard (`#include <iostream>`). Alcuni ambienti di compilazione più vecchi, in particolare versioni non recenti di DJGPP, non riconoscono questa notazione, per cui sarà necessario aggiungere un `.h` al nome del file da includere (`#include <iostream.h>`).

[1] lunedì 1º marzo, ore 9:30-11:30, aula A23 (2L)

Ripasso degli array, relazione array - puntatore

D: 9.1, 9.2; C: 7.6

La memoria in un elaboratore è organizzata come un vettore lineare di celle, ognuna con un indirizzo. Una variabile in un linguaggio è associata ad un'area di memoria o ad un registro della CPU. Un puntatore `p` ad una variabile `a` è a sua volta una variabile che contiene l'indirizzo di `a`.

Cosa succede quando dobbiamo allocare un gran numero di variabili? Usiamo un array, che ha un solo nome e ai cui elementi si accede usando un indice. La notazione nome+indice significa base+scostamento, ed è equivalente ad usare i puntatori. In C/C++ questa equivalenza è resa esplicita usando gli opportuni costrutti, infatti `a[b]` è *esattamente equivalente* a `*(a+b)`.

Le stringhe

D: 9.6

In C le stringhe sono viste come array di caratteri *ASCII*, di lunghezza pari alla lunghezza della stringa più 1, perché all'ultimo posto si mette uno 0. Questa particolare codifica è detta *ASCIIZ*, e presenta particolari caratteristiche di efficienza nella manipolazione delle stringhe. È questo uno dei punti di forza del C.

Esempio: funzioni `length`, `strcmp`

Facciamo degli esempi che mostrano come il C può essere estremamente conciso ed estremamente vicino alla macchina. Queste considerazioni valgono in generale per C e C++. Conviene usare le caratteristiche di basso livello del linguaggio solo se non si è sicuri della qualità del compilatore e si hanno effettivamente esigenze di efficienza del codice. In tutti gli altri casi, la stragrande maggioranza, è preferibile usare costrutti chiari piuttosto che efficienti. Sempre per ragioni di chiarezza, ed anche per facilitare il debugging, è spesso preferibile utilizzare indici piuttosto che puntatori.

Le funzioni di manipolazione delle stringhe della libreria standard del C sono molto usate, e per questa ragione particolarmente ottimizzate. Implementiamo la funzione `strlen` in tre modi diversi: accedendo ai caratteri della stringa usando una notazione ad array con indice, usando i puntatori, e infine ottimizzando il più possibile. Da notare che il carattere ASCII 0 è indicato con `'\0'`.

Sorgente: strfun.cc

```
int
array_strlen (const char *s)
{
  int i;
  for (i = 0; s[i] != '\0'; i++)
    continue;
  return i;
}

int
pointer_strlen (const char *s)
```

```
{
  const char *t = s;
  while (*t != '\0')
    t += 1;
  return t-s;
}

int
strlen (const char *s)
{
  const char *t = s;
  while (*t++);
  return t-s-1;
}
```

Esaminiamo ora la funzione `strcmp`. Anche qui vediamo una implementazione che utilizza gli indici, una che utilizza i puntatori, ed una che cerca di ottenere la massima efficienza.

Sorgente: strfun.cc

```
int
array_strcmp (const char *s1, const char *s2)
{
  int i;
  for (i = 0; s1[i] != '\0' && s1[i] == s2[i]; i++)
    continue;
  return s1[i] - s2[i];
}

int
pointer_strcmp (const char *s1, const char *s2)
{
  while (*s1 != '\0' && *s1 == *s2)
    {
      s1 += 1;
      s2 += 1;
    }
  return *s1 - *s2;
}

int
strcmp (const char *s1, const char *s2)
{
  while (*s1)
    if (*s1++ != *s2++)
      return *--s1 - *--s2;
  return *s1 - *s2;
}
```

[2] martedì 2 marzo, ore 17:00-18:00, aula A32 (1E)

Esercizio: funzione `strncmp`

Funziona come la `strcmp`, ma prende un argomento in più, che indica il massimo numero di caratteri da confrontare.

Sorgente: strfun.cc

```
int
array_strncmp (const char *s1, const char *s2, int n)
{
  for (int i = 0; i < n; i++)
    if (s1[i] == '\0' || s1[i] != s2[i])
      return s1[i] - s2[i];
  return 0;
}

int
pointer_strncmp (const char *s1, const char *s2, int n)
{
  while (n > 0)
    if (*s1 != '\0' && *s1 == *s2)
      {
        n -= 1; s1 += 1; s2 += 1;
      }
    else
      return *s1 - *s2;
  return 0;
}
```

Esempio: mappatura dei caratteri in un altro insieme, `inline`

C: 11.5

Vogliamo implementare la codifica crittografica che Cesare usava per le proprie comunicazioni. Si tratta di un semplice sistema che consiste nel sostituire ad ogni carattere alfabetico il carattere che viene due posti dopo. Per esempio, il carattere `'c'` viene convertito nel carattere `'e'`, ed il carattere `'Z'` nel carattere `'B'`. Eccone un'implementazione semplice:

Sorgente: caesar.cc

```
char
caesar_simple (char c)
{
  if (c >= 'A' && c <= 'Z')
    return 'A' + (c-'A'+2)%26;
  else if (c >= 'a' && c <= 'z')
    return 'a' + (c-'a'+2)%26;
  else return c;
}
```

uu

Esiste un modo semplice e molto più efficiente di ottenere lo stesso risultato, a prezzo di chiamare una funzione di inizializzazione. In compenso, la funzione di traduzione è estremamente efficiente.

Sorgente: caesar.cc

```
static char caesar_table [256];

void
init_caesar_table ()
{
  for (int i = 0; i < 256; i++)
    caesar_table[i] = i;
  for (int i = 0; i < 26; i++)
    {
      caesar_table['A'+i] = 'A'+(i+2)%26;
      caesar_table['a'+i] = 'a'+(i+2)%26;
    }
}

char
caesar (char c)
{
  return caesar_table[c];
}
```

[3] mercoledì 3 marzo, ore 14:00-16:00, aula A32 (1L,1E)

Esercizio: funzione `toupper`

Utilizzando una tabella di traduzione come nell'esempio precedente, si può implementare in maniera efficiente la funzione della libreria C che serve a convertire un carattere alfabetico in maiuscolo. I caratteri non alfabetici rimangono invariati.

Sorgente: toupper.cc

```
char toupper_table [256];

void
init_toupper_table ()
{
  for (int i = 0; i < 256; i++)
    toupper_table[i] = i;
  for (int i = 0; i < 26; i++)
    toupper_table['a'+i] = 'A'+i;
}

inline char
toupper (char c)
{
  return toupper_table[c];
}
```

Per provare la funzione vista è opportuno creare un programmino che legga una stringa da input e ne scriva la traduzione:

Sorgente: toupper.cc

```
#include <iostream>

char *toupper_string (char *s)
{
  for (char *t = s; *t != '\0'; t++)
    *t = toupper(*t);
  return s;
}

int main ()
{
  char stringa [100];

  init_toupper_table();

  cout << "Immetti una stringa: ";
  cin >> stringa;
  cout << stringa << endl;
  cout << toupper_string(stringa) << endl;

  return 0;
}
```

Ripasso di I/O da cin/cout

D: 5.1, 5.4

L'output formattato in C si effettua usando la funzione `printf`, che è molto flessibile. In C++ l'uso dell'operatore `<<` permette di scrivere rapidamente semplici programmi usa e getta, ma è altrettanto complesso di `printf` quando si vuole un controllo preciso sull'output.

Sorgente: printf.cc

```
#include <iostream>
#include <iomanip>
#include <stdio.h>

main ()
{
  int i = 12345678;
  double d = 12345678.1234567;
  char *s = "1234567890";

  // Le formattazioni semplici si fanno meglio in C++
  printf("i = %d\n", i);
  cout << "i = " << i << endl;
  cout << endl;

  // Anche quelle appena piu' complesse
  printf("%s %d %g\n", s, i, d);
```

```
  cout << s << ' ' << i << ' ' << d << endl;
  cout << endl;

  // Quelle complicate si fanno piu' succintamente con printf
  cout << setw(20) << s
       << "   " << "0x" << hex << i
       << setw(15) << setprecision(10) << d << endl;
  printf("%20s   0x%x%15.10g\n", s, i, d);

  return 0;
}

i = 12345678
i = 12345678

1234567890 12345678 1.23457e+07
1234567890 12345678 1.23457e+07

          1234567890   0xbc614e    12345678.12
          1234567890   0xbc614e    12345678.12
```

Esempio: riempire un array con dati da standard input

D: 5.2, 5.3

Sorgente: arrayio.cc

```
#include <iostream>
#include <fstream>

const char *asciiname = "asciidata";
const int N = 5;
int v [N];

bool
input ()
{
  for (int i = 0; i < N; i++)
    {
      cout << "v[" << i << "] <-- ";
      if (!(cin >> v[i]))
        return false;
    }

  return true;
}
```

[4] lunedì 8 marzo, ore 9:30-11:30, aula A23 (2L)

Bit field, operazioni sulle maschere di bit

D: 3.2.1

Maschere logiche, operazioni sui bit, operatori not, and, or, shift aritmetico e logico, operazioni set bit, clear bit, set mask, clear mask. Bit field in C come maniera più pulita di effettuare operazioni di mascheramento. Una volta erano spesso usati per risparmiare spazio, oggi si usano soprattutto per riflettere una configurazione hardware.

```
// accendere il bit n-esimo della variabile a, lunga L bit
a |= 1 << n;              // n compreso fra 0 e L-1

// spegnere il bit n-esimo
a &= ~(1 << n);           // n compreso fra 0 e L-1
```

Esempio: controllore per una lavatrice industriale

Supponiamo di scrivere il programma di controllo *embedded* per una lavatrice industriale. Il programma accede in lettura e scrittura ad un *porta* o *registro* (una locazione di memoria) di una lavatrice industriale che segnala i sensori attivati e la temperatura, e pilota la resistenza.

Sorgente: resisten.cc

```
// extern significa che la variabile e‘ dichiarata in un altro file.
// volatile che il suo contenuto puo‘ cambiare autonomamente, ed il
//   compilatore ne deve tener conto.
// unsigned che le operazioni di shift su di essa devono essere logiche,
//   non aritmetiche.
// short, su tutte le architetture esistenti, significa 16 bit, ma in
//   teoria e‘ dipendente dal compilatore.

// quando il registro e‘ 0 la macchina non e‘ funzionante
extern volatile unsigned short reg;

const unsigned short ON =       0x8000; // tasto di accensione
const unsigned short EMERG =    0x4000; // interruttore di emergenza
const unsigned short RESIST =   0x1000; // attuatore resistenza
const unsigned short TEMP_D =   0x03e0; // temperatura desiderata
const unsigned short TEMP_M =   0x001f; // sensore di temperatura

void
gestione_resistenza ()
{
  for (;;) if (reg != 0)
    {
      if (reg & ON
          && !(reg & EMERG)
          && (reg & TEMP_M) < ((reg & TEMP_D) >> 5))
        reg |= RESIST;
```

```
      else
        reg &= ~RESIST;
    }
}
```

[5] martedì 9 marzo, ore 17:00-18:00, aula A32 (1L)

Espressioni logiche, trasformazioni di espressioni booleane

D: 3.4; C: 2.2.2

Ancora sulle operazioni a bit: operatore xor, operazioni toggle, set all, clear all, estrazione di un campo di m bit alla posizione n.

```
// estrarre un campo di m bit posto alla posizione n della var. a
unsigned int a;

unsigned int res;
res = (1 << m) - 1;                 // creo un maschera di m bit
res <<= n;                          // la sposto di n posti a sinistra
res &= a;                           // estraggo i bit interessanti da a
res >>= n;                          // e li porto in posizione utilizzabile
```

And e or come prodotto e somma nell'algebra booleana a due valori. Tabelle di verità. Connettori logici in C, il tipo bool e i valori true e false, gli operatori and, or, not. Il valore logico di un'espressione numerica è falso se l'espressione ha valore 0, vero altrimenti. Trasformazione di un'arbitraria espressione logica nella sua negata. Implicazione: se rispondo al telefono allora sono in casa; questa affermazione è falsa solo se rispondo al telefono senza essere in casa.

```
(x > 0 && x < y)                    //  0 < x < y
(x <= 0 || x >= y)                  //  negazione di 0 < x < y
(!(a && !b))                        //  a implica b
(!a || b)                           //  a implica b
(b != 0 && a % b == 0)              //  a divisibile per b
```

[6] mercoledì 10 marzo, ore 14:00-16:00, aula A32 (2E)

Esercizio: controllo di antifurto auto

Si abbia un controllore di antifurto per auto. Il controllore contiene un registro con dei bit associati ad interruttori, altri a sensori e dei bit associati agli attuatori. Gli interruttori sono l'interruttore di accensione dell'antifurto ed due ponticelli che determinano la durata dell'allarme: 0 vuol dire infinito, 1 vuol dire 30 secondi, 2 vuol dire 1 minuto, 3 vuol dire 5 minuti. I sensori sono i segnali di accensione e spegnimento provenienti dal telecomando ed i vari sensori di scasso dell'auto. Il telecomando è costituito di due pulsanti, ognuno dei quali emette un segnale diverso. I due pulsant sono indipendenti fra di loro ed ognuno di essi emette un segnale fintantoché è premuti. Gli attuatori sono la sirena di allarme e l'alimentazione della benzina. Esiste inoltre un registro contasecondi accessibile in lettura, il cui valore incrementa di uno ogni secondo, e che si azzera quando è acceduto in scrittura. Ecco la definizione dei registri e dei bit:

Sorgente: allarme.cc

```
extern volatile short reg;      // registro sensori e attuatori

// Ogni sensore e' ad 1 quando e' attivo
unsigned short ON = 0x8000;     // interruttore di accensione
unsigned short DUR_SZ = 2;      // dimensione del codice di durata
unsigned short DUR_POS = 8;     // posizione del codice di durata
unsigned short TEL_ON = 0x0800; // il telecomando segnala accensione
unsigned short TEL_OFF = 0x0400; // il telecomando segnala spegnimento
unsigned short RUOTE = 0x0080;  // allarme ruote
unsigned short MOV = 0x0040;    // allarme movimento
unsigned short SPORT = 0x0020;  // allarme apertura sportello
unsigned short VOL = 0x0010;    // allarme sensore volumetrico
unsigned short SIRENA = 0x0008; // accensione sirena (1 suona)
unsigned short POMPA = 0x0004;  // pompa benzina (1 funziona)

// il timer si azzera quando vi si scrive qualunque cosa
extern volatile short timer;    // orologio con avanzamento a secondi

void
controlla_sensori_mask ()
{
  bool acceso = false;

  while (true)
    {
      while (!acceso)
        {
          acceso = (reg & ON
                    && reg & TEL_ON
                    && !(reg & TEL_OFF));
        }
      while (acceso)
        {
          if (!(reg & ON)
              || reg & TEL_OFF)
            {
              reg &= ~SIRENA;
              reg |= POMPA;
              acceso = false;
            }
          else if (reg & (RUOTE | MOV | SPORT | VOL))
            {
              reg |= SIRENA;
              reg &= ~POMPA;
              unsigned short mask = ((1 << DUR_SZ) - 1) << DUR_POS;
              int codice = (reg & mask) >> DUR_POS;
              if (codice != 0)
                {
                  int limite [] = { 0, 30, 60, 300 };
                  timer = 0;
                  while (timer < limite[codice]
                         && reg & ON
                         && !(reg & TEL_OFF))
                    continue;
                  reg &= ~SIRENA;
```

```
                reg |= POMPA;
              }
            } /* else if (uno dei sensori attivo) */
        } /* while (acceso) */
    } /* while (true) */
}

// Stesso programma implementato con i bit field
extern volatile struct
{
  int on               :1;     // 0x8000
  int                  :1;     // 0x4000
  int dur              :2;     // 0x3000
  int tel_on           :1;     // 0x0800
  int tel_off          :1;     // 0x0400
  int                  :2;     // 0x0300
  int ruote            :1;     // 0x0080
  int mov              :1;     // 0x0040
  int sport            :1;     // 0x0020
  int vol              :1;     // 0x0010
  int sirena           :1;     // 0x0008
  int pompa            :1;     // 0x0004
  int                  :2;     // 0x0003
} r;    // registro sensori e attuatori

void
controlla_sensori_bit_field ()
{
  bool acceso = false;

  while (true)
    {
      while (!acceso)
        {
          acceso = (r.on && r.tel_on && !r.tel_off);
        }
      while (acceso)
        {
          if (!r.on || r.tel_off)
            {
              r.sirena = 0;
              r.pompa = 1;
              acceso = false;
            }
          else if (r.ruote || r.mov || r.sport || r.vol)
            {
              r.sirena = 1;
              r.pompa = 0;
              if (r.dur != 0)
                {
                  int limite [] = { 0, 30, 60, 300 };
                  timer = 0;
                  while (timer < limite[r.dur]
                         && r.on
                         && !r.tel_off)
                    continue;
                  r.sirena = 0;
```

```
                r.pompa = 1;
              }
            } /* else if (uno dei sensori attivo) */
        } /* while (acceso) */
    } /* while (true) */
}
```

Spiegare in pseudo codice una logica ragionevole per l'antifurto. Esempio di soluzione:

```
stato = spento;
while (true)
{
  while (stato == spento)
  {
    if (interruttore_di_accensione
        && accensione_dal_telecomando
        && !spegnimento_dal_telecomando)
      stato = acceso;
  }
  while (stato == acceso)
  {
    if (!interruttore_di_accensione
        || spegnimento_dal_telecomando)
      {
        spegni sirena e attiva pompa benzina;
        stato = spento;
      }
    else if (uno dei sensori attivo)
      {
        accendi sirena e disattiva pompa benzina;
        if (durata != 0)
        {
          limite = durata in secondi;
          azzera il contasecondi;
          while (contasecondi < limite
                 && interruttore_di_accensione
                 && !spegnimento_dal_telecomando)
            continue;
          spegni sirena e attiva pompa benzina;
        }
      } /* else if (uno dei sensori attivo) */
  } /* while (stato == acceso)
} /* while (true) */
```

Scrivere una funzione che, dentro un loop infinito, implementi la logica descritta. Esempio di soluzione con mascheramento e shift:

Sorgente: allarme.cc

```
void
controlla_sensori_mask ()
{
  bool acceso = false;

  while (true)
    {
```

```
        while (!acceso)
          {
            acceso = (reg & ON
                      && reg & TEL_ON
                      && !(reg & TEL_OFF));
          }
        while (acceso)
          {
            if (!(reg & ON)
                || reg & TEL_OFF)
              {
                reg &= ~SIRENA;
                reg |= POMPA;
                acceso = false;
              }
            else if (reg & (RUOTE | MOV | SPORT | VOL))
              {
                reg |= SIRENA;
                reg &= ~POMPA;
                unsigned short mask = ((1 << DUR_SZ) - 1) << DUR_POS;
                int codice = (reg | mask) >> DUR_POS;
                if (codice != 0)
                  {
                    int limite [] = { 0, 30, 60, 300 };
                    timer = 0;
                    while (timer < limite[codice]
                           && reg & ON
                           && !(reg & TEL_OFF))
                      continue;
                    reg &= ~SIRENA;
                    reg |= POMPA;
                  }
              } /* else if (uno dei sensori attivo) */
          } /* while (acceso) */
      } /* while (true) */
}
```

Esempio di soluzione con i bit field:

Sorgente: allarme.cc

```
extern volatile struct
{
  int on                :1;     // 0x8000
  int                   :1;     // 0x4000
  int dur               :2;     // 0x3000
  int tel_on            :1;     // 0x0800
  int tel_off           :1;     // 0x0400
  int                   :2;     // 0x0300
  int ruote             :1;     // 0x0080
  int mov               :1;     // 0x0040
  int sport             :1;     // 0x0020
  int vol               :1;     // 0x0010
  int sirena            :1;     // 0x0008
  int pompa             :1;     // 0x0004
  int                   :2;     // 0x0003
} r;    // registro sensori e attuatori
```

```
void
controlla_sensori_bit_field ()
{
  bool acceso = false;

  while (true)
    {
      while (!acceso)
        {
          acceso = (r.on && r.tel_on && !r.tel_off);
        }
      while (acceso)
        {
          if (!r.on || r.tel_off)
            {
              r.sirena = 0;
              r.pompa = 1;
              acceso = false;
            }
          else if (r.ruote || r.mov || r.sport || r.vol)
            {
              r.sirena = 1;
              r.pompa = 0;
              if (r.dur != 0)
                {
                  int limite [] = { 0, 30, 60, 300 };
                  timer = 0;
                  while (timer < limite[r.dur]
                         && r.on
                         && !r.tel_off)
                    continue;
                  r.sirena = 0;
                  r.pompa = 1;
                }
            } /* else if (uno dei sensori attivo) */
        } /* while (acceso) */
    } /* while (true) */
}
```

L’esempio visto finora può essere lo stimolo per una serie di esercizi possibili, in quanto la logica usata per gestire l’antifurto non è l’unica ragionevole. Per fare un esempio, si potrebbero mettere dei ritardi di un secondo sulla pressione dei tasti del telecomando, in modo che debbano essere premuti per almeno un secondo prima di essere presi in considerazione. Inoltre si può mettere un ritardo di cinque secondi al sensore di spertura dello sportello, che permetta al proprietario che ha smarrito il telecomando di entrare in macchina e disattivare l’antifurto dall’interruttore di accensione senza far suonare l’allarme.

[7] lunedì 15 marzo, ore 9:30-11:30, aula A23 (1L,1E)

Operazioni su dati sequenziali: i filtri

Uso di filtri in un sistema operativo. Ammettiamo di avere dei dati numerici in ASCII, uno per riga, in un file, e di voler contare quanti sono pari ad un dato valore x. In Unix si fa:

```
cat file | grep x | wc -l
```

Gli esempi di filtri che faremo saranno indirizzati ad operazioni su file ASCII contenenti un numero per riga. File di questo genere possono essere prodotti da un programma che legge uno strumento di misura, o da un simulatore, o essere il risultato di un'elaborazione effettuata con programmi di altro tipo.

Esempio: calcolare il quadrato

Costruiamo un filtro che legge sullo standard input una sequenza di numeri in virgola mobile in formato ASCII separati da spazi o fine riga, e per ognuno di essi ne scrive sullo standard output il quadrato, sempre in formato ASCII, seguito da un fine riga.

Sorgente: f-sqr.cc

```
#include <iostream>

int
main ()
{
  double n;

  while (cin >> n)
    if (!(cout << n*n << '\n'))
      {
        cerr << "Errore di scrittura su stdout" << endl;
        return 1;
      }

  return 0;
}
```

Esercizio: calcolare la media

Costruiamo ora un filtro che legge sullo standard input una sequenza ASCII di numeri, e scriva sullo standard output un singolo numero seguito da fine riga, pari alla media dei numeri in input.

Sorgente: f-mean.cc

```
#include <iostream>

int
main ()
{
  double n;
  double somma = 0;
  unsigned long quanti = 0;

  while (cin >> n)
```

```
    {
      somma += n;
      quanti += 1;
    }
  if (quanti > 0)
    if (!(cout << somma/quanti << '\n'))
      {
        cerr << "Errore di scrittura su stdout" << endl;
        return 1;
      }

  return 0;
}
```

Usando i due filtri appena costruiti si può calcolare il valor quadratico medio di una serie di dati. Questa è una misura molto usata in elettronica, ed è appunto pari alla media del quadrato di una quantità variabile nel tempo. Ecco come si calcolerebbe in Unix il valor quadratico medio dei valori numerici presenti in un file chiamato `misure`:

```
cat misure | f-sqr | f-mean
```

Ed ecco come si farebbe in DOS:

```
type misure | f-sqr | f-mean
```

Come esempio del frequente utilizzo del valor quadratico medio di una grandezza, cito il *valore efficace*, che è la radice quadrata del valor quadratico medio di una grandezza variabile nel tempo. Ad esempio, il valore comunemente noto della tensione alternata domestica, che è di 220V, è in termini tecnici il valore efficace di quella tensione, che varia nel tempo con l'andamento di una sinusoide.

Esempio: compressione ed espansione della dinamica

Costruiamo un filtro che operi la seguente trasformazione sui valori di input, essendo *x* un valore di input ed *y* il corrispondente valore di output:

```
y = sqrt(x)     per x >= 0
y = -sqrt(-x)   per x < 0
```

La precedente trasformazione è un esempio di *compressione della dinamica* di un segnale. La trasformazione inversa ripristina il segnale originale, ed è detta *espansione della dinamica*:

```
y = x*x         per x >= 0
y = -x*x        per x < 0
```

Ecco un'implementazione dei due filtri.

Sorgente: f-compr.cc

```
#include <iostream>
#include <cmath>          // usare <math.h> per i vecchi compilatori

int
main ()
```

```
{
  double n;

  while (cin >> n)
    {
      if (n >= 0)
        n = sqrt(n);
      else
        n = -sqrt(-n);
      if (!(cout << n << '\n'))
        {
          cerr << "Errore di scrittura su stdout" << endl;
          return 1;
        }
    }

  return 0;
}
```

Sorgente: f-espand.cc

```
#include <iostream>
#include <cmath>           // usare <math.h> per i vecchi compilatori

int
main ()
{
  double n;

  while (cin >> n)
    {
      if (n >= 0)
        n = n*n;
      else
        n = -n*n;
      if (!(cout << n << '\n'))
        {
          cerr << "Errore di scrittura su stdout" << endl;
          return 1;
        }
    }

  return 0;
}
```

La compressione ed espansione della dinamica è un'operazione solitamente utilizzata per elaborare segnali acustici. Ammettiamo di leggere il segnale registrato su di un CD, e di volerlo registrare su di un nastro magnetico. La dinamica di un CD è molto maggiore di quella di un nastro magnetico, per cui la registrazione taglierebbe i suoni più forti o, se riduciamo il volume, perderebbe i più deboli fra il rumore di fondo del nastro. Una soluzione consiste nel comprimere la dinamica prima di registrare sul nastro, e di riespanderla prima di riprodurla, quando si legge il nastro.

Siccome i due filtri sono uno l'inverso dell'altro, se messi in cascata non cambiano l'input. Il filtro costituito dalla composizione dei due scrive sullo standard output gli stessi numeri che legge sullo standard input, a meno di errori di imprecisione numerica:

```
f-compr | f-espans
```

[8] lunedì 15 marzo, ore 16:00-17:00, aula AI1 (1S)

Laboratorio: l'ambiente di sviluppo RHIDE

Introduzione all'ambiente di sviluppo libero RHIDE, editor, compilatore ed ambiente di esecuzione. Esempio di compilazione ed esecuzione del sorgente printf.cc.

[9] mercoledì 17 marzo, ore 14:00-16:00, aula A32 (2E)

Esercizio: media esponenziale

Implementare il filtro che, per ogni numero x_i letti in input scriva in output un numero y_i così definito:

$$y_i = \alpha y_{i-1} + (1-\alpha)x_i,$$
$$y_1 = x_1,$$
$$0 < \alpha < 1.$$

Ecco una soluzione:

Sorgente: f-exmean.cc

```
#include <iostream>

const double alfa = 0.9;

int
main ()
{
  double x, y;

  if (cin >> x)
    y = x;                  // inizializza y0
  do
    {
      y = alfa*y + (1-alfa)*x;
      cout << y << '\n';
      if (!cout)
        {
          cerr << "Errore di scrittura su stdout" << endl;
          return 1;
        }
```

```
    } while (cin >> x);

  return 0;
}
```

La funzione implementata è spesso detta *media esponenziale* con *fattore di decadimento* α.

Esercizio: sottocampionamento con passo fisso

Ammettiamo di avere una lista di misurazioni di una grandezza fisica espresse come valori numerici, e di volerne ottenere un estratto avente un minor grado di dettaglio (*sottocampionamento*). Per esempio, potremmo avere un sensore termometrico che fornisce una rilevazione della temperatura cento volte al secondo, ma noi siamo solo interessati ad averne una al minuto. Oppure, abbiamo della musica campionata su CD con 44000 campioni al secondo, e vogliamo ottenerne una registrazione di minore qualità e minore ingombro, accontentandoci di 22000 campioni al secondo.

Implementare il filtro che legga dall'input una sequenza di numeri separati da blank e ne scriva in output uno ogni N. Più precisamente, data una sequenza x_i in input, il filtro scriverà sull'output la sequenza così definita:

$$y_i = x_{Ni} .$$

Questo è un esempio di soluzione:

Sorgente: f-camp.cc

```
#include <iostream>

static int N = 10;              // scrive un numero ogni 10

int
main ()
{
  double n;
  int conto = 0;

  while (cin >> n)
    {
      conto += 1;
      if (conto == N)
        {
          cout << n << '\n';
          conto = 0;
        }
    }

  return 0;
}
```

Esercizio: sottocampionamento con media

Esiste un modo più naturale per effettuare un sottocampionamento di una serie di dati. Invece di prendere un numero ogni N, consideriamo blocchi di N numeri e prendiamo la loro media.

Implementare il filtro che legga dall'input una sequenza di numeri separati da blank e ne scriva in output uno ogni N, pari alla media degli ultimi N letti. Più precisamente, data una sequenza x_i in input, il filtro scriverà sull'output la sequenza così definita:

$$y_i = (x_{N(i-1)+1} + x_{N(i-1)+2} + \ldots + x_{N(i-1)+N}) / N.$$

Questo è un esempio di soluzione:

Sorgente: f-campm.cc

```
#include <iostream>

static int N = 10;              // scrive un numero ogni 10

int
main ()
{
  double n;
  double somma = 0;
  int conto = 0;

  while (cin >> n)
    {
      somma += n;
      conto += 1;
      if (conto == N)
        {
          cout << somma/N << '\n';
          somma = 0;
          conto = 0;
        }
    }

  return 0;
}
```

[10] lunedì 22 marzo, ore 9:30-10:30, aula A23 (1L)

Dispatch table, tabella con voci di menù (comando, puntatore a funzione e stringa di commento)

D: 11.1, 11.2

I puntatori a funzione sono utili ogniqualvolta si debba chiamare uuna funzione senza sapere a priori quale sarà chiamata. Si tratta di un meccanismo a basso livello, utilizzabile anche in C. Usiamolo per costruire un programma che presenta un menù dal quale scegliere possibili operazioni logiche sui bit da effettuare su un intero. La scelta di quale funzione chiamare è basata su di un carattere che viene introdotto dall'utente, ed è usato come indice in una tabella di puntatori a funzione.

Una tale tabella è comunemente nota come *dispatch table* (tabella di chiamata). Si tratta di un metodo molto efficiente per decidere quale funzione eseguire. Un sistema analogo è utilizzato dai compilatori per tradurre il costrutto `switch` del linguaggio C. Il compilatore può costruire una tabella in ogni voce corrisponde ad uno dei possibili valori della variabile di `switch`. Ogni voce dell'array corrispondente ad uno dei `case` contiene l'indirizzo al quale il programma salta per quel valore della variabile. In questo caso, si parla più comunemente di *jump table*.

La dispatch table è riempita a partire da un'altra tabella, che descrive i possibili comandi. La tabella di descrizione dei comandi è implementata come un array di struct, dove ogni voce della tabella (o *record*) è una struct costituita di tre *campi*. Il primo campo è il carattere che l'utente deve immettere per dare il comando, il secondo campo è un puntatore alla funzione che esegue il comando, il terzo è un puntatore ad un array di caratteri (una stringa) che contiene una breve descrizione del comando.

`typedef`

D: 12.4

Per semplificare la lettura, usiamo una `typedef` per creare il tipo «puntatore a funzione con un argomento `unsigned` che restituisce un `unsigned`». La `typedef` crea un sinonimo per il nome di un tipo.

Sorgente: bit-menu.h

```
typedef unsigned int (*command_func) (unsigned int);

struct voce_comandi {
  char input;
  command_func funzione;
  char *nome;
};
```

[11] mercoledì 24 marzo, ore 15:00-16:00, aula A32 (1L)

Tempo di vita e visibilità delle variabili

D: 14.3, 14.3.1

Il *tempo di vita* di una variabile è il tempo di esecuzione del programma durante il quale quella variabile esiste, cioè può essere utilizzata per assegnarle o leggerne un valore. Distinguiamo perciò le variabili in *statiche* ed *automatiche*.

Sono *statiche* le *variabili globali*, cioè definite al di fuori delle funzioni, e le *variabili locali* alle funzioni precedute dal *qualificatore* `static`. Tali variabili esistono fin dall'inizio del programma, prima che sia eseguita la prima istruzione della funzione `main`, e non smettono mai di esistere fino all'uscita dal programma. Esse sono inizializzate a 0 prima dell'inizio del programma. Gli array hanno tutti i loro elementi inizializzati a 0, e le struct hanno tutti i loro membri inizializzati a 0.

Sono *automatiche* le variabili locali alle funzioni che non siano precedute dal qualificatore `static`. Tali variabili sono definite solo durante l'esecuzione della funzione a cui sono locali. Prima che la funzione sia chiamata, o dopo che essa sia uscita con un `return`, le variabili non esistono. Questo implica che il valore assegnato ad una varibile automatica è perduto una volta che la funzione sia uscita. Le variabili automatiche non sono inizializzate automaticamente: il valore di una variabile automatica quando una funzione viene chiamata è un numero casuale, che generalmente varia cambiando il sistema operativo, il compilatore, o semplicemente facendo girare più volte lo stesso programma.

La *visibilità* (*scope*) di una variabile è l'insieme dei posti di un programma da cui la variabile può essere usata. In C/C++ la visibilità di una variabile globale è l'intero programma, quello di una variabile locale ad una funzione è la funzione in cui è definita. È anche possibile restringere ad un singolo file sorgente la visibilità di una variabile esterna alle funzioni, facendola precedere dal qualificatore `static`. *Tale uso del qualificatore* `static` *è completamente diverso dall'uso prima descritto*. Il qualificatore `static` può anche essere usato per le funzioni. In tal caso, la funzione può essere chiamata solo da funzioni definite nello stesso file sorgente.

Inizializzazione implicita ed esplicita delle variabili

D: 9.1, 10.1…

È possibile inizializzare un array ed al tempo stesso definirne la dimensione implicitamente. È anche possibile inizializzare una struct. Sono possibili tutti i casi di annidamento, risolti con l'uso opportuno delle parentesi graffe.

Sorgente: bit-func.cc

```
voce_comandi comandi [] =
{
  { '?', help, "mostra l'elenco dei comandi" },
  { 'i', input_number, "inserisci un numero" },
  { '!', logical_not, "not logico" },
  { '<', shift_left, "shift a sinistra" },
  { '>', logical_shift_right, "shift a destra logico" },
  { '/', arithmetic_shift_right, "shift a destra aritmetico" },
  { '\0' }
};
```

[12] lunedì 29 marzo, ore 9:30-11:30, aula A23 (1L,1E)

Le `union`

D: 10.2…

Una union serve per poter utilizzare in modi diversi una stessa area di memoria. In un programma portabile, se si usa un campo della union in scrittura, si deve usare lo stesso campo in lettura.

Ammettiamo ad esempio di dover leggere dei sensori e registrare in un array gli istanti in cui uno di questi cambia valore ed il nuovo valore. Abbiamo due sensori, accessibili attraverso la lettura di due porte hardware, `sa` ed `sb`. La lettura di `sa` fornisce un valore intero, mentre la lettura di `sb` fornisce un numero in virgola mobile. Abbiamo inoltre una porta hardware `timer`, che contiene un valore intero che misura il tempo in una qualche unità di misura. Utilizziamo una struttura per registrare una misura. La struttura contiene una lettura del timer nell'istante in cui il valore è registrato, un codice che indica quale valore è registrato, ed il valore stesso:

```
extern unsigned int timer;
extern int sa;
extern float sb;

enum tipo_sensore { sens_a, sens_b };

struct lettura
{
  unsigned int time;
  tipo_sensore sensore;
  union {
    int int_v;
    float float_v;
  } valore;
};

const int N = 1000;
lettura misure [N];

void
effettua_letture ()
{
  int ultimo_sa = sa;
  float ultimo_sb = sb;
  int indice = 0;

  for (;;)
    {
      if (sa != ultimo_sa)
        {
          ultimo_sa = sa;
          indice += 1;
          misure[indice].time = timer;
          misure[indice].sensore = sens_a;
          misure[indice].valore.int_v = sa;
        }
      if (sb != ultimo_sb)
```

```
          {
            ultimo_sb = sb;
            indice += 1;
            misure[indice].time = timer;
            misure[indice].sensore = sens_b;
            misure[indice].valore.float_v = sb;
          }
      }
  }
```

L'esempio, per brevità, comprende due soli sensori, ma la convenienza d'uso della union in termini di risparmio di memoria diventa chiara se si considera che lo stesso criterio può essere adottato per un numero arbitrario di sensori.

Mentre il codice precedente è compilabile anche da compilatori C, il C++ consente un'abbreviazione della sintassi, che è molto comoda in casi del genere, mediante l'uso delle *union anonime*:

Sorgente: union-ex.cc

```
extern unsigned int timer;
extern int sa;
extern float sb;

enum tipo_sensore { sens_a, sens_b };

struct lettura
{
  unsigned int time;
  tipo_sensore sensore;
  union {
    int int_v;
    float float_v;
  };
};

const int N = 1000;
lettura misure [N];

void
effettua_letture ()
{
  int ultimo_sa = sa;
  float ultimo_sb = sb;
  int indice = 0;

  for (;;)
    {
      if (sa != ultimo_sa)
        {
          ultimo_sa = sa;
          indice += 1;
          misure[indice].time = timer;
          misure[indice].sensore = sens_a;
          misure[indice].int_v = sa;
        }
      if (sb != ultimo_sb)
```

```
            {
              ultimo_sb = sb;
              indice += 1;
              misure[indice].time = timer;
              misure[indice].sensore = sens_b;
              misure[indice].float_v = sb;
            }
        }
    }
```

Le `struct` in memoria, endianness

Il linguaggio C/C++ non specifica come il compilatore debba allocare in memoria i membri di una struttura in qualche modo particolare. L'unico vincolo è che l'ordine in memoria sia lo stesso in cui i membri sono dichiarati all'interno della struttura. In generale, una struttura in memoria può avere dei «buchi», cioè delle aree di memoria inutilizzate. Nella maggior parte delle architetture, per evitare i buchi, è sufficiente ordinare i membri in ordine decrescente di lunghezza.

Sfruttiamo ora il costrutto union in una maniera non portabile, appositamente, scrivendo su un campo e leggendo da un altro. In particolare, in questa union, il risultato della lettura di `c` dopo aver scritto su `uint` è dipendente dalla *endianness* della macchina, cioè dal fatto che la macchina abbia ordinamento *big endian* o *little endian*.

```
union input_number
{
  unsigned int uint;
  unsigned char c [4];
} x;
```

Funzione `exit`

D: 7.10

La funzione `exit` si usa per uscire dal programma da una funzione che non sia la `main`, da dove si esce normalmente con un `return`. L'argomento della `exit` viene pasasto al sistema operativo, esattamente nello stesso modo del valore ritornato dalla funzione `main`.

Sorgente multifile

D: 14.2.2, 14.2.3

Il programma è organizzato in tre file. Uno di essi è un *header file*, che contiene solo *dichiarazioni* di variabili e funzioni esterne e di tipi, cioè `typedef`, `struct` e `union`. Questo file è incluso dagli altri due. Il primo di questi contiene il main e delle funzioni accessorie. È un file pensato per non essere modificato spesso, perché contiene il corpo del funzionamento del programma. L'ultimo file contiene la tabella delle voci del menù e le funzioni che implementano i comandi. Ogni volta che si vuol modificare o aggiungere un nuovo comando, solo questo file va modificato.

Sorgente: bit-menu.h

```
typedef unsigned int (*command_func) (unsigned int);

struct voce_comandi {
  char input;
  command_func funzione;
  char *nome;
};

extern voce_comandi comandi [];
extern command_func tabella [];
```

Sorgente: bit-menu.cc

```
#include <iostream>
#include <iomanip>
#include "bit-menu.h"

char *
bin_print (unsigned int n)
{
  const int N = 9*sizeof(int);
  // s deve essere static, perche' viene restituito il suo indirizzo, e
  // quindi deve ancora esistere dopo che la funzione esce.  s[N]=='\0'
  // perche' tutte le variabili static sono inizializzate a 0.
  static char s [N+1];

  for (int i = 0; i < N; i++)
    {
      if (i % 9 == 0)
        {
          s[i] = ' ';
          continue;
        }
      if (n & 0x80000000)
        s[i] = '|';
      else
        s[i] = 'o';
      n <<= 1;
    }

  return s;
}

void
output_with_iostream (unsigned int in)
{
  union input_number
  {
    unsigned int uint;
    unsigned char c [4];
  } x;

  x.uint = in;

  // Scrivi il numero nell'ordine in cui i byte sono allocati in memoria,
  // dall'indirizzo piu' basso al piu' alto.
  cout << hex << setfill('0') << "  0x"
```

```
        << setw(2) << (int)x.c[0]
        << setw(2) << (int)x.c[1]
        << setw(2) << (int)x.c[2]
        << setw(2) << (int)x.c[3] << '\n';

  // Assegna il valore a delle variabili piu' piccole: lo short e' lungo 2
  // byte e il char e' lungo 1 byte.
  short sshort = x.uint;
  unsigned short ushort = x.uint;
  char schar = x.uint;
  unsigned char uchar = x.uint;
  cout << hex << setfill('0')
       << "  0x" << setw(8) << x.uint
       << "      0x" << setw(4) << (int)ushort
       << "        0x" << setw(2) << (int)uchar
       << "   " << bin_print(x.uint) << '\n'
       << dec << setfill(' ');
  cout << setw(12) << x.uint
       << setw(12) << (unsigned int)ushort
       << setw(12) << (unsigned int)uchar << '\n';
  cout << setw(12) << (int)x.uint
       << setw(12) << sshort
       << setw(12) << (int)schar << '\n';
}

int
main ()
{
  // Verifica che questo compilatore si comporti come previsto.
  if (sizeof(short) != 2 || sizeof(int) != 4)
    {
      cerr << "Le dimensioni di short o int non sono quelle attese\n";
      return 1;
    }

  // Inizializza la tabella di salto
  int i;
  for (i = 0; comandi[i].input != '\0'; i++)
    tabella[comandi[i].input] = comandi[i].funzione;

  // Inizializza l'elenco dei comandi disponibili
  char elenco_comandi [256];
  for (i = 0; comandi[i].input != '\0'; i++)
    elenco_comandi[i] = comandi[i].input;
  elenco_comandi[i] = '\0';

  // Leggi un carattere di input ed esegui il comando corrispondente.
  char c;
  unsigned int x;
  cin.unsetf(ios::dec);
  while (cout << "Comandi (" << elenco_comandi << ") ? "
         && cin >> c)
    {
      if (tabella[c] == NULL)
        cout << "  Il comando \"" << c << "\" non esiste\n";
      else
        x = tabella[c](x);
```

```
        output_with_iostream(x);  // stampa in diversi formati
      }

  return 0;
}
```

Sorgente: bit-func.cc

```
#include <iostream>
#include <stdlib.h>
#include "bit-menu.h"

unsigned int input_number (unsigned int);
unsigned int help (unsigned int);
unsigned int logical_not (unsigned int);
unsigned int shift_left (unsigned int);
unsigned int logical_shift_right (unsigned int);
unsigned int arithmetic_shift_right (unsigned int);
unsigned int and_mask (unsigned int);

voce_comandi comandi [] =
{
  { '?', help, "mostra l'elenco dei comandi" },
  { 'i', input_number, "inserisci un numero" },
  { '!', logical_not, "not logico" },
  { '<', shift_left, "shift a sinistra" },
  { '>', logical_shift_right, "shift a destra logico" },
  { '/', arithmetic_shift_right, "shift a destra aritmetico" },
  { '&', and_mask, "and con maschera da input" },
  { '\0' }
};

command_func tabella [256];

unsigned int
help (unsigned int unused)
{
  cout << "Comandi disponibili:\n";
  for (int i = 0; comandi[i].input != '\0'; i++)
    cout << ' ' << comandi[i].input << '\t' << comandi[i].nome << '\n';
  return unused;
}

unsigned int
input_number (unsigned int discarded)
{
  unsigned int x;
  if (!(cout << "Numero (oct con 0, hex con 0x, dec altrimenti)? "
        && cin >> x))
    {
      cerr << "Errore di input\n";
      exit(1);
    }
  return x;
}
```

```
unsigned int
logical_not (unsigned int x)
{
  return !x;
}

unsigned int
shift_left (unsigned int x)
{
  return x << 1;
}

unsigned int
logical_shift_right (unsigned int x)
{
  return x >> 1;
}

unsigned int
arithmetic_shift_right (unsigned int x)
{
  return (int)x >> 1;
}

unsigned int
and_mask (unsigned int x)
{
  return x & input_number(0);   // l'argomento a input_number e' ignorato
}
```

Media mobile

La *media mobile con finestra di lunghezza N* è un'operazione che si effettua su dati sequenziali, che serve ad eliminare il rumore o i picchi stretti. Per ogni numero in ingresso a partire dall'*N*-esimo, se ne genera uno in uscita che sia la media degli ultimi N. Formalmente:

$$y_i = (x_{i-N+1} + x_{i-N+2} + \ldots + x_i) / N.$$

Ecco un filtro che implementa la media con finestra mobile di lunghezza *N*:

Sorgente: f-mobil1.cc

```
#include <iostream>

const int N = 10;               // scrive un numero ogni 10

int
main ()
{
  double ultimi[N];
  int conto = 1;

  while (cin >> ultimi[N-1])
    {
      if (conto < N)
```

```
        conto += 1;
      else
        {
          double somma = 0;
          for (int i = 0; i < N; i++)
            somma += ultimi[i];
          cout << somma/N << '\n';
        }
      for (int i = 0; i < N-1; i++)
        ultimi[i] = ultimi[i+1];
    }

  return 0;
}
```

Esempio: media mobile con un anello implementato con un array

Sorgente: f-mobil2.cc

```
#include <iostream>

const int N = 10;               // scrive un numero ogni 10

int
main ()
{
  double ultimi[N] = { 0 };
  double x;
  int indice = 0;
  double somma = 0;
  bool full = false;

  while (cin >> x)
    {
      somma += x;
      ultimi[indice] = x;
      full = full || (indice == N-1);
      if (full)
        cout << somma/N << '\n';
      indice = (indice+1) % N;
      somma -= ultimi[indice];
    }

  return 0;
}
```

[13] lunedì 29 marzo, ore 16:00-17:00, aula AI1 (1S)

Laboratorio: le operazioni logiche sui bit

Esempio di esecuzione del programma `bit-menu`, con esemplificazione del funzionamento delle operazioni sui bit e delle conversioni fra int di diversa lunghezza e signed/unsigned. Aggiunta della funzione «not bit a bit» al programma `bit-menu`.

[1º compitino] martedì 30 marzo, ore 17:10, aula B21

[14] mercoledì 31 marzo, ore 14:00-16:00, aula A32 (2L)

I/O su file in ASCII

D: 5.5

È frequente il caso di avere dei dati numerici generati da un primo programma ed elaborati da un secondo. Specialmente se i programmi sono sviluppati in tempi diversi o da persone diverse, e non è possibile accordarsi su di un formato comune per i dati, conviene utilizzare il formato di dati più portabile, che è l'ASCII. Con questa scelta, si ha fra l'altro il vantaggio che il contenuto dei file di dati può essere ispezionato con un qualunque editor. Anche usando l'ASCII, tuttaia possono esistere problemi di compatibilità fra sistemi operativi diversi. Infatti il carattere di fine riga è `\n` in Unix, `\r\n` su DOS e derivati, `\r` su Mac. La convenzione di usare `\n` per indicare il fine riga è nata in Unix. Le librerie dei compilatori per DOS e Mac effettuano automaticamente la conversione nella convenzione locale.

Esempio: scrivere l'array su file in ASCII, rileggerlo, stamparlo

Sorgente: arrayio.cc

```
bool
scrivi_rileggi_e_stampa ()
{
  fstream datafile(asciiname, ios::out|ios::trunc);
  int i;

  if (!datafile)
    return false;

  for (i = 0; i < N; i++)
    if (!(datafile << v[i] << '\n'))
      return false;

  datafile.close();

  datafile.open(asciiname, ios::in);

  cout << endl;
  for (i = 0; i < N; i++)
    {
      int n;
```

```
        if (datafile >> n)
          cout << "  v[" << i << "] = " << setw(5) << n;
        else
          return false;
      }
    cout << endl;

    return true;
  }
```

Esempio: scrivere in binario

C: 11.2.3.4

Il formato ASCII per memorizzare su file i dati ha il vantaggio della portabilità e della facile leggibilità senza programmi appositi, ma è spesso poco compatto, e richiede maggior tempo di elaborazione per la lettura e la scrittura. Se è richiesta la massima efficienza, a scapito di portabilità e facilità di ispezione, il formato binario è il più adeguato, a patto che sia precisamente specificato. L'ordinamento dei byte può causare problemi di portabilità nel passaggio fra macchine con diversa endianness. Attenzione ad aprire il file in modo binario, a meno di non scrivere programmi che verranno usati esclusivamente su Unix, per evitare la conversione automatica del fine riga. In ogni caso, leggere la documentazione delle librerie che si usano.

Sorgente: arrayio.cc

```
const char *binname = "bindata";

bool
scrivi_in_binario ()
{
  fstream datafile(binname, ios::bin|ios::out|ios::trunc);
  for (int i = 0; i < N; i++)
    if (!(datafile.write(&v[i], sizeof(v[i]))))
      return false;

  return true;
}
```

[15] lunedì 12 aprile, ore 9:30-11:30, aula A23 (2L)

Panoramica sull'I/O su file in C e in C++

parti di D: 5; C: 6.2.2.3, parti di 11

In C++ si usa la libreria iostream, mentre in C si usa la libreria stdio. Come tutte le librerie del C, quest'ultima può essere utilizzata anche in C++, ma in generale non si possono mischiare nello stesso programma operazioni su iostream con operazioni su stdio.

Uso di iostream per standard input/output:

```
#include <iostream>            // uso consigliato dallo standard C++
#include <iostream.h>          // alternativa per vecchi compilatori

int a;
cin >> a;

// Le seguenti tre forme sono esattamente equivalenti,
// perche‘ << e‘ un operatore binario associativo a sinistra
// che restituisce un riferimento al suo operando sinistro.
cout << a << ’ ’ << setw(8) << 5.5 << "  fine\n";
((((cout << a) << ’ ’) << setw(8)) << 5.5) << "  fine\n";
cout << a; cout << ’ ’; cout << setw(8); cout << 5.5; cout << "  fine\n";
```

Uso di stdio per standard input/output:

```
scanf(" %d", &a);
printf("%d %8g  fine\n", a, 5.5);
```

Uso di iostream per operazioni sui file:

```
#include <fstream>             // uso consigliato dallo standard C++
#include <fstream.h>           // alternativa per vecchi compilatori

fstream f;                             // variabile di tipo fstream
f.open("nome", ios::in);               // apri il file in lettura
f.open("nome", ios::out|ios::trunc); // apri il file in scrittura

fstream f("nome", ios::in);            // in alternativa
fstream f("nome", ios::out|ios::trunc); // in alternativa

if (!f) error();                       // controllo dell’errore

f >> a;                                // input
f << a;                                // output

f.close()                              // chiusura
```

Uso di stdio per operazioni sui file:

```
#include <stdio.h>

FILE *f;                               // variabile di tipo puntatore a FILE
f = fopen("nome", O_RDONLY);           // apri il file in lettura
f = fopen("nome", O_WRONLY|O_TRUNC); // apri il file in scrittura

if (f == NULL) error();                // controllo dell’errore dopo open

fscanf(f, " %d", &a);                  // input
fprintf(f, " %d", a);                  // output

if (ferror(f)) error();                // controllo dell’errore
if (feof(f)) fine_file();              // controllo fine file

fclose(f)                              // chiusura
```

Esercizio: leggere array in binario da file, stamparlo in ordine inverso

Sorgente: arrayio.cc

```
bool
leggi_e_stampa_inverso ()
{
  fstream datafile(binname, ios::bin|ios::in);
  int i;

  for (i = 0; i < N; i++)
    if (!(datafile.read(&v[i], sizeof(v[i]))))
      return false;

  for (i = N-1; i >= 0; i--)
    cout << "  v[" << i << "] = " << setw(5) << v[i];
  cout << endl;

  return true;
}

int
main ()
{
  if (input()
      && scrivi_rileggi_e_stampa()
      && scrivi_in_binario()
      && leggi_e_stampa_inverso())
    return 0;

  cerr << "Errore di i/o\n";
  return 1;
}
```

I riferimenti in C++

D: 8.3

Un *riferimento* in C++ permette di riferirsi ad un oggetto con un nome diverso:

```
int i;
int& ri = i;                // int& e‘ il tipo di ri
const int& cri = i;         // qui il tipo e‘ const int&

ri = 5;                     // ora i e‘ uguale a 5, ri non cambia
cri = 5;                    // questo e‘ un errore
```

Si possono usare al posto dei puntatori come argomenti delle funzioni, ma non è una pratica consigliabile.

```
void scambiap (int *pa, int *pb)
{
  int t = *pa; *pa = *pb; *pb = t;
}

void scambiar (int& ra, int& rb)
{
  int t = ra; ra = rb; rb = t;
}
```

La cosa interessante è che possono essere utilizzati anche come valore di ritorno delle funzioni. In questo esempio, usare dei riferimenti come argomenti è necessario, e non sarebbe possibile usare dei puntatori.

```
int& minr (int& ra, int& rb)
{
  if (ra < rb) return ra;
  else return rb;
}

int a, b;
…
minr(a,b) = 0;
```

Filtri a carattere, funzione `get`

C: 11.2.3.2; D: 20.5

Il modo più efficiente per leggere da un `iostream` un carattere alla volta è usare la funzione `get`. Ecco un programma che funziona da filtro, leggendo lo standard input ed applicandovi la codifica di Cesare precedentemente implementata nel file caesar.cc. Il modulo che segue va collegato assieme a **`caesar.cc`** per ottenere un file eseguibile.

Sorgente: f-caesar.cc

```
#include <iostream>

extern void init_caesar_table ();
extern char caesar (char);

int
main ()
{
  init_caesar_table();
  char c;
  while (cin.get(c))
    cout << caesar(c);
  return 0;
}
```

La funzione `get` può anche essere chiamata senza argomenti. In tal caso, essa restituisce un `int`, che è uguale alla costante `EOF` quando si legge oltre la fine dello stream.

Analogamente, in C, esiste la funzione `getc (FILE f)`, che opera su uno stream aperto in lettura, e restituisce un `int` contenente il carattere letto o `EOF`. La funzione `getchar()` è equivalente a `getc(stdin)`.

Array di stringhe, `argc`, `argv`.

Un array di stringhe è un array di puntatori a carattere. Ogni puntatore punta ad una zona in memoria dove è memorizzata la stringa in codifica ASCIIZ. Quando un programma in C/C++ viene eseguito, questo conosce il suo *ambiente* di esecuzione (execution *environment*) per mezzo di tre argomenti alla funzione `main`. Il prototipo completo della funzione è:

```
int main (int argc, char **argv, char **env);
```

Si noti che è legale omettere gli ultimi argomenti della funzione, per cui sono legali gli altri due stili comunemente usati per la funzione `main`:

```
int main ();
int main (int argc, char **argv);
```

`argc` è il numero di argomenti passati al programma, che normalmente è maggiore o uguale ad 1. `argv` è un array di stringhe. Il primo elemento dell'array `argv`, cioè la stringa `argv[0]` è il nome con cui il programma è stato chiamato. Gli elementi successivi sono gli argomenti passati al programma: `argv[1]` è il primo argomento, `argv[2]` è il secondo, e così via. Analogamente, `env` è un array di stringhe, che contiene le *variabili d'ambiente* impostate dal sistema operativo. Il loro numero non è noto mediante un contatore, come nel caso di `argc` e `argv`, ma il puntatore successivo all'ultima stringa è un puntatore nullo.

[16] lunedì 12 aprile, ore 16:00-17:00, aula AI1 (1S)

Laboratorio: i filtri, introduzione al debugger

Esempio di esecuzione dei filtri `f-sqr` ed `f-mean`. Uso di RHIDE per seguire l'esecuzione di un programma, visualizzazione delle variabili.

[17] mercoledì 14 aprile, ore 14:00-16:00, aula A32 (1L,1E)

Esercizio: dump esadecimale di un file binario

È frequente che un programma che scrive i suoi risultati sullo standard output accetti come argomento il nome di un file da usare come input, ed usi invece lo standard input se non ha argomenti, comportandosi in tal modo da filtro. Scriviamo un programma che si comporti come descritto, che scriva sullo standard output una rappresentazione esadecimale byte per byte del proprio input:

Sorgente: hexdump.cc

```
#include <iostream>
#include <fstream>
#include <iomanip>

bool
read_and_print_from (istream& ins)
{
  int c;
  for (int i = 0; i < 8; i++)
    if ((c = ins.get()) != EOF)
      cout << ' ' << setw(2) << setfill('0') << hex << c;
  return !ins;
}

int main (int argc, char **argv)
{
  istream *infp;
  if (argc < 2)
  {
    // mi comporto da filtro e leggo lo standard input
    infp = &cin;
    cout << "Dump esadecimale da standard input:\n";
  }
  else
  {
    // apro il file il cui nome e' il primo argomento
    // static serve perche' altrimenti infp punterebbe ad una variabile
    //   automatica, che non esisterebbe piu' fuori dal blocco else
    static ifstream infile(argv[1], ios::bin|ios::in);
    if (infile)
    {
      // nessun errore in apertura: usa infile per la lettura
      infp = &infile;
      cout << "Dump esadecimale del file \"" << argv[1] << "\"\n";
    }
    else
    {
      cerr << "Errore di apertura del file " << argv[1] << endl;
      return 1;
    }
  }

  bool finito;
  bool finelinea = false;
  do
  {
    finito = read_and_print_from(*infp);
    if (finito || finelinea)
      cout << '\n';
    else
      cout << ' ';
    finelinea = !finelinea;
```

```
  } while (!finito);

  return 0;
}
```

Array a più dimensioni

D: 9.5.2

La dichiarazione di variabile `int a [4]` alloca un array `a` di quattro elementi di tipo `int`. Analogamente, la dichiarazione di variabile `int b [5] [4]` alloca un array `b` di cinque elementi di tipo *array di quattro int*. Quando si tratta di una dichiarazione `extern`, o del parametro formale di una funzione, è possibile omettere la prima dimensione, così:

```
extern int a [];
extern int b [] [4];
extern int c [] [5] [4];
double f (double m []);
double g (double m [] [4]);
double h (double m [] [5] [4]);
```

L'array a due dimensioni `b` è allocato in memoria come una sequenza di cinque array di quattro int, e rappresenta l'oggetto matematico *matrice di cinque righe e quattro colonne*. Ecco la rappresentazione matematica:

00	01	02	03
10	11	12	13
20	21	22	23
30	31	32	33
40	41	42	43

e l'organizzazione in memoria sequenziale di `b`:

00	01	02	03	10	11	12	13	20	21	22	23	30	31	32	33	40	41	42	43

Passaggio di matrici come argomenti di funzioni

C: 9.4.4

La funzione `g` sopra definita ha come argomento un array a due dimensioni formato di un numero indefinito di elementi, ognuno dei quali è un array di quattro interi. Se all'interno della funzione si accede ad un elemento come `b[i][j]`, il compilatore lo interpreta come accesso all'elemento di ordine i*4+j. Si noti come in questa espressione il compilatore deve conoscere gli indici usati per l'accesso (`i` e `j`) ed il numero di colonne, ma non ha bisogno di conoscere il numero di righe.

Esempio: funzioni somma, prodotto

Vogliamo costruire una libreria di funzioni matematiche che operino sulle matrici. Le funzioni che vogliamo scrivere devono poter funzionare per matrici di qualunque, quindi non possiamo usare il metodo di passaggio dei parametri sopra descritto. Questo significa che, se vogliamo ad esempio scrivere una funzione che sommi due matrici, mettendo il risultato nella prima, il prototipo della funzione sarebbe:

```
void somma (double risultato [] [C], addendo [] [C], int r);
```

Quindi potremmo scrivere una funzione che opera su matrici con numero arbitrario di righe `R`, ma con numero fisso di colonne `C`. Per scrivere una funzione che sia utilizzabile con matrici con arbitrario numero di colonne possiamo fare come nella funzione `somma`, `prodotto`, o `stampa`, mentre la funzione `prodotto3` agisce su matrici di dimensione fissa 3x3:

Sorgente: matrix.cc

```
#include <stdio.h>

// Somma le matrici RISULTATO e ADDENDO, e poni la somma in
// RISULTATO.  Tutte le matrici hanno dimensione RxC.
void
somma (double *risultato, double *addendo, int R, int C)
{
  for (int i = 0; i < R; i++)
    for (int j = 0; j < C; j++)
      risultato[i*C+j] += addendo[i*C+j];
  // In questo caso particolare, si potrebbe scrivere piu' semplicemente:
  // for (int i = 0; i < C*R; i++)
  //   risultato[i] += addendo[i];
}

// Moltiplica le matrici A e B e poni il prodotto in RISULTATO.
// Tutte le matrici hanno dimensione 3x3.
void
prodotto3 (double risultato[3][3], double a[3][3], double b[3][3])
{
 for (int i = 0; i < 3; i++)
   for (int j = 0; j < 3; j++)
   {
     double res = 0;
     for (int k = 0; k < 3; k++)
       res += a[i][k] * b[k][j];
     risultato[i][j] = res;
   }
}

// Moltiplica le matrici A e B e poni il prodotto in RISULTATO.
// Tutte le matrici hanno dimensione NxN.
void
prodotto (double *risultato, double *a, double *b, int N)
{
 for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
   {
```

```
      double res = 0;
      for (int k = 0; k < N; k++)
        res += a[i*N+k] * b[k*N+j];
      risultato[i*N+j] = res;
    }
}

// Stampa su cout la matrice A, di dimensione RxC.
void
stampa (char *nome, double *a, int R, int C)
{
  printf("Matrice %s (%dx%d):\n", nome, R, C);
  for (int i = 0; i < R; i++)
  {
    for (int j = 0; j < C; j++)
      printf("%12g", a[i*C+j]);
    printf("\n");
  }
}

int
main ()
{
  double m1 [3] [3] = { {1,2,3}, {2,3,4}, {3,4,5} };
  double m2 [3] [3] = { {1,1,1}, {1,1,1}, {1,1,1} };
  double m3 [3] [3];

  stampa("m1", (double *)m1, 3, 3);
  stampa("m2", (double *)m2, 3, 3);

  somma((double *)m1, (double *)m2, 3, 3);
  stampa("m1 += m2", (double *)m1, 3, 3);

  prodotto3(m3, m1, m2);
  stampa("m3 = m1 x m2", (double *)m3, 3, 3);

  prodotto((double *)m3, (double *)m1, (double *)m2, 3);
  stampa("m3 = m1 x m2", (double *)m3, 3, 3);

  return 0;
}
```

[18] lunedì 19 aprile, ore 9:30-11:30, aula A23 (2L)

Allocazione dinamica: malloc e free, new e delete

D: 13.1; C: 17.1

Spesso avviene che la dimensione di un array può essere conosciuta solo a runtime (durante l’esecuzione). Per esempio, scriviamo un filtro che legga dei numeri da standard input per poi stamparli in ordine inverso. Il programma deve mettere i numeri letti in un array e poi rileggerlo dalla fine all’inizio, così:

```
#include <stdio>
int
{
  main ()
  int a [1000];
  int i;
  for (i = 0; i < 1000; i++)
    if (!(cin >> a[i]))
      break;
  for (i -= 1; i >= 0; i--)
    cout << a[i];
  return 0;
}
```

In questo caso abbiamo scelto una dimensione fissa per l'array, cioè 1000 elementi. In generale questo non va bene, perché potremmo avere più di 1000 numeri da leggere e stampare. Se sappiamo qual è la dimensione della memoria della macchina, possiamo usare un array che la occupi tutta, in maniera da ottenere la massima prestazione possibile, ma questo vorrà dire che il nostro programma non potrà girare su una macchina più piccola, e non sfrutterà la memoria di una macchina più grossa. Inoltre, in un sistema multitasking, il nostro programma occuperà sempre tutta la memoria, sottraendola agli altri processi, anche per leggere e stampare pochi numeri.

È possibile fare in modo che un programma occupi solo la memoria che gli serve, e che questa decisione sia presa durante l'esecuzione, e non a tempo di compilazione. Il concetto di riservare della memoria durante l'esecuzione si chiama *allocazione dinamica* della memoria. La memoria allocata durante la compilazione si dice *allocata staticamente*.

Il filtro **`f-revers`** esemplifica questi concetti: il programma legge dai numeri in formato ASCII dallo standard input e li scrive in ordine inverso sullo standard output, utilizzando uno di tre metodi alternativi per l'allocazione dinamica.

Il primo metodo è utilizzato quando **`f-revers`** viene chiamato con un argomento numerico positivo. In tal caso, **`f-revers`** alloca un array di lunghezza pari al numero dato come argomento e legge dall'input fino alla fine dell'input o fino a riempire l'array.

Il secondo metodo è utilizzato quando il primo argomento sulla linea di comando di **`f-revers`** comincia col carattere **`m`**. In questo caso, si alloca con la funzione `malloc`, che fa parte della libreria standard del C (e del C++), un array di una dimensione iniziale arbitraria prefissata (piccola). Se ci sono in input più numeri di quanti ne entrano nell'array, si usa la `realloc` per ingrandire l'area di memoria allocata, e così via finchè finisce l'input o la memoria.

Il terzo metodo è utilizzato quando il primo argomento sulla linea di comando di **`f-revers`** comincia col carattere **`n`**. In questo caso, si alloca con l'operatore del C++ `new` un array di una dimensione iniziale arbitraria prefissata (piccola). Se ci sono in input più numeri di quanti ne entrano nell'array, si alloca un altro array di dimensione maggiore sempre usando la `new`, ci si copia il contenuto del vecchio array e si dealloca il vecchio array usando l'operatore del C++ `delete []`. Il concetto è esattamente lo stesso del caso precedente, ma se si usano la `new` e la `delete`, non esiste un equivalente della `realloc`, quindi la copia va fatta esplicitamente.

Sorgente: f-revers.cc

```
#include <iostream>
#include <stdlib.h>

double *a;

// Errore di invocazione
void
arg_error ()
{
  cerr << "Il filtro si aspetta un argomento: se questo e' un numero,\n"
       << "usera' la routine read_n allocando il numero dato di elementi\n"
       << "altrimenti si aspettera' il carattere m (malloc) o n (new).\n";
  exit (1);
}

// Leggi al piu' n numeri
int
read_n (int n)
{
  a = new double [n];
  int i;
  for (i = 0; i < n; i++)
    if (!(cin >> a[i]))
      break;                    // uscita dal ciclo
  return i;
}

// Leggi finche' c'e' memoria (allocata con malloc)
int
read_malloc ()
{
  const int min = 1;            // in un programma vero e' piu' grande, almeno 100
  int i = 0;
  int n = min;
  a = (double *)malloc(n * sizeof(*a));
  while (a != NULL)
    {
      for (; i < n; i++)
        if(!(cin >> a[i]))
          return i;             // uscita dal ciclo e dalla funzione
      n *= 2;
      a = (double *)realloc(a, n*sizeof(*a));
    }
  cerr << "Memoria esaurita\n";
  exit(1);
}

// Leggi finche' c'e' memoria (allocata con new)
int
read_new ()
{
  const int min = 1;            // in un programma vero e' piu' grande, almeno 100
  int i = 0;
  int n = min;
  a = new double [n];
```

```
  while (true)                  // niente check su a: la libreria genera un errore
    {
      for (; i < n; i++)
        if (!(cin >> a[i]))
          return i;             // uscita dal ciclo e dalla funzione
      double *tmp = new double [2*n];
      for (int j = 0; j < n; j++)
        tmp[j] = a[j];          // non esiste l'equivalente della realloc
      delete [] a;
      a = tmp;
      n *= 2;
    }
}

// Vedi la routine arg_error per il funzionamento.
int
main (int argc, char *argv[])
{
  if (argc != 2)
    arg_error ();

  int n;
  switch (argv[1][0])
    {
    case 'm':
      n = read_malloc();
      break;
    case 'n':
      n = read_new();
      break;
    case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
      n = read_n(atoi(argv[1]));
      break;
    default:
      arg_error();
    }

  for (n -= 1; n >= 0; n--)
    cout << a[n] << '\n';

  return 0;
}
```

Matrice a due dimensioni come array di puntatori

Un modo generico ed efficiente di costruire ed usare matrici a due dimensioni consiste nell'implementare la matrice bidimensionale come un array di puntatori ad array. Il codice che segue costruisce una tale struttura:

Sorgente: matrix2.cc

```
typedef double **matrice;

// Crea una matrice RxC costruita come un array di R puntatori
// a vettori di lunghezza C, e restituisce la base dell'array
```

```
// di puntatori a vettori.
matrice
crea_matrice (int r, int c)
{
  matrice mat = new (double *) [r];
  for (r--; r >= 0; r--)
    mat[r] = new double [c];
  return mat;
}
```

Questo modo di implementare una matrice bidimensionale è molto interessante per due motivi. Il primo è che consente di accedere agli elementi della matrice usando esattamente la stessa sintassi che si usa nel caso di array a due dimensioni, e cioè la notazione `a[i][j]`. Questa notazione funziona perchè `a[i]` accede al puntatore all'i-esimo vettore (che contiene l'i-esima riga), e `a[i][j]` accede al j-esimo elemento in quel vettore, quindi alla j-esima colonna.

Il secondo motivo per cui questa implementazione è interessante è che essa è più efficiente rispetto ad usare un array bidimensionale, perchè non è necessario effettuare nessun calcolo, implicito o esplicito, per stabilire la posizione in memoria dell'elemento acceduto, ma solo due indirezioni, cioè non sono necessarie una moltiplicazione, una somma, un'indirezione ed un accesso in memoria, ma due indirezioni e due accessi in memoria. Normalmente la seconda alternativa è più veloce.

Un sicuro svantaggio di questa implementazione rispetto all'array bidimensionale è che la struttura occupa più spazio in memoria, e che per allocarla e deallocarla è necessario un ciclo `for` appositamente scritto. Ecco la funzione che dealloca una `matrice` come su definita:

Sorgente: matrix2.cc

```
// Dealloca una matrice allocata usando crea_matrice.  Se questa
// crea_matrice e distruggi_matrice sono riscritte usando malloc e free,
// distruggi_matrice deve prendere due argomenti come crea_matrice.
void
distruggi_matrice (matrice a, int r)
{
  for (r--; r >= 0; r--)
    delete [] a[r];
  delete [] a;
}
```

[19] lunedì 19 aprile, ore 16:00-17:00, aula AI1 (1S)

Laboratorio: argomenti ed ambiente di un programma, uso del debugger

Il seguente programma stampa il valore di `argc`, i propri argomenti, ed il proprio ambiente.

Sorgente: printenv.cc

```
#include <iostream>

int
main (int argc, char **argv, char **env)
{
  cout << "argc = " << argc << '\n' << endl;

  for (int i = 0; i < argc; i++)
    cout << "argv[" << i << "] = \"" << argv[i] << '"' << endl;
  cout << endl;

  for (char **s = env; *s != NULL; s++)
    cout << '"' << *s << '"' << endl;
  return 0;
}
```

Compilazione, esecuzione, tracciamento e modifica del sorgente f-caesar.cc. Breakpoint semplici. Come funziona lo svuotamento del buffer dello stream di output.

[20] mercoledì 21 aprile, ore 14:00-16:00, aula A32 (2E)

Esercizio: filtro FIR con coefficienti dati

Un filtro FIR è un algoritmo matematico che prende una sequenza di numeri x_i ne ricava una sequenza di numeri y_i così calcolata:

$$y_n = a_o x_{n-k} + a_1 x_{n-k+1} + \ldots + a_k x_n$$

Dove i parametri $a_o \ldots a_k$ sono dei coefficienti dati, detti appunto i coefficienti del filtro. Questo algoritmo si può vedere come una generalizzazione dell'algoritmo della media mobile, che abbiamo già visto. In effetti, un filtro che implementi una media mobile con finestra di lunghezza *k* è un filtro FIR con $a_o = 1/k \ldots a_{k-1} = 1/k$.

Implementare un programma che funzioni da filtro FIR, leggendo i valori x_i dallo standard input e scrivendo i valori y_i sullo standard output. I valori in input e in output siano in virgola mobile, in formato ASCII separati da blank. I coefficienti a_j siano dati come argomenti sulla linea di comando: il primo sia a_o, il secondo a_1 e così via.

Sorgente: fir.cc

```
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <stdio.h>

void
errore ()
{
  cerr << "Gli argomenti devono essere numeri\n";
```

```
  exit (1);
}

main (int argc, char *argv[])
{
  // Leggo gli argomenti sulla linea di comando e li metto in a
  int n = argc - 1;
  if (n < 1)
    errore ();
  float *a = new float [n];
  for (int i = 0; i < n; i++)
    {
      // a[i] = atof(argv[i+1]);
      int quanti = sscanf(argv[i+1], "%f", &a[i]);
      if (quanti != 1)
        errore ();
    }

  // Uso un contatore per leggere i primi n-1 numeri senza stampare
  double *x = new double [n];
  int in = 1;
  while (cin >> x[n-1])
    {
      if (in < n)
        in += 1;
      else
        {
          double out = 0;
          for (int i = 0; i < n; i++)
            out += a[i] * x[i];
          // Manca il controllo di errore sull'output
          cout << out << '\n';
        }
      for (int i = 1; i < n; i++)
        x[i-1] = x[i];
    }
  return 0;
}
```

Esercizio: distribuzione di una serie di dati

Supponiamo di avere una sequenza di dati e di voler veder come sono distribuiti. Il metodo consiste nello scegliere un valore massimo e minimo fra i quali interessa calcolare la distribuzione, e di un numero di categorie (*bin*) in cui dividere questo intervallo. Quindi, per ogni numero della sequenza di dati, determinare a quale categoria appartiene. Alla fine del processo, il risultato è il numero di dati che si trovano in ciascuna categoria.

Implementare un programma che legga da file in binario una sequenza di numeri in virgola mobile in doppia precisione, e scriva sullo standard output una sequenza di interi positivi in formato ASCII, uno per linea. Ogni intero rappresenta il numeri di dati di input che si trova in una categoria. Il programma prende due parametri sulla riga di comando: il primo è il nome del file di input, il secondo è il numero di categorie *n*. La categoria 0 ospita i numeri da 0 a 1 escluso, la categoria 1 ospita i numeri da 1 a 2 escluso, e così via. I numeri inferiori a 0 o maggiori o uguali a *n* sono scartati.

Sorgente: distrib.cc

```
#include <iostream>
#include <fstream>
#include <stdlib.h>

void
errore ()
{
  cerr << "Errore negli argomenti\n";
  exit (1);
}

int
main (int argc, char *argv[])
{
  if (argc != 3)
    errore ();
  char *filename = argv[1];
  int n = atoi(argv[2]);

  int *cat = new int [n];
  for (int i = 0; i < n; i++)
    cat[i] = 0;
  ifstream f (filename, ios::bin|ios::bin);
  if (!f)
    errore ();

  double x;
  while (f.read(&x, sizeof(x)))
    if (x >=0 && x < n)
      cat[(int)x] += 1;

  for (int i = 0; i < n; i++)
    cout << cat[i] << '\n';
  return 0;
}
```

[21] lunedì 26 aprile, ore 9:30-11:30, aula A23 (2L)

Ricerca lineare e binaria

D: 15.4.1, 15.4.2

Supponiamo di avere un array di elementi, e di doverne cercare uno. In generale, il modo più ragionevole è di partire dall'inizio e, per ogni elemento, confrontarlo con quello cercato fino a trovarlo, o a non trovarlo, dopo aver scorso tutti gli elementi. Se gli elementi sono sistemati a caso nell'array, il numero medio di confronti è pari a metà del numero di elementi, e nel caso pessimo, in cui l'elemento ricercato non si trova, il numero di confronti è pari al numero degli elementi.

Ad esempio, ammettiamo di avere un array di record anagrafici, del tipo:

```
struct anagrafico {
  char nome [20];
  char cognome [40];
  int annonascita;
  enum { M, F } sesso;
}
```

e di voler cercare un dato cognome. Useremo la funzione `strncmp` per confrontare il cognome cercato con quelli presenti nell'array. L'algoritmo descritto si chiama *ricerca lineare*. Se il numero di elementi nell'array è *N*, il numero medio di confronti è pari a 1/2 *N*, mentre il numero pessimo di confronti è *N*. Sia il caso medio che quello pessimo, quindi, sono proporzionali ad *N* secondo una costante. Si dice che l'algoritmo ha complessità O(*N*) nel numero di elementi presenti nell'array. Ecco un pezzo di codice di esempio:

Sorgente: ricerca.cc

```
#include <string.h>

// In tutte le funzioni, l ed r sono gli indici minimo e massimo dell'array
// passato come argomento.  Per effettuare la ricerca su tutti gli elementi
// di un array di lunghezza N bisognera' passare gli indici 0 e N-1.

int
ricerca_lineare (char *dato, anagrafico a[], int l, int r)
{
  for (int i = l; i <= r; i++)
    if (!strncmp(a[i].cognome, dato, 40))
      return i;
  return -1;
}
```

Supponiamo ora di sapere che l'array è ordinato, cioè che i record sono sistemati con i cognomi in ordine alfabetico. Il miglior metodo di ricerca consiste allora in qualcosa di simile a quel che si fa cercando un nome nell'elenco del telefono: si guarda l'elemento centrale, se è minore di quello cercato si guarda l'elemento centrale della metà superiore, se è maggiore si guarda l'elemento centrale della metà inferiore e così via. In media, bastano $\log_2(N)$ confronti. Questo algoritmo si chiama *ricerca binaria*, ed ha complessità O(log*N*).

Sorgente: ricerca.cc

```
int
ricerca_binaria (char *dato, anagrafico a[], int l, int r)
{
  while (l <= r)
    {
      int m = (l+r)/2;
      int cfr = strncmp(a[m].cognome, dato, 40);
      if (cfr == 0)
        return m;
      if (cfr > 0)
        r = m-1;
```

```
      else
        l = m+1;
    }
  return -1;
}
```

Selection sort

Un semplice algoritmo per effettuare l'ordinamento di un array consiste nello scegliere l'elemento più piccolo (selezione) e sostituirlo al primo elemento dell'array. Quindi effettuare la stessa operazione per gli elementi dell'array dal secondo in poi e così via. Eccone un'implementazione:

Sorgente: sort.cc

```
struct data
{
  int key;
  int data;
};

void
scambia (data& a, data& b)
{
  data tmp = a; a = b; b = tmp;
}

void
selection (data a[], int n)
{
  for (int i = 0; i < n-1; i++)
    {
      int min = i;
      for (int j = i+i; j < n; j++)
        if (a[j].key < a[min].key)
          min = j;
      scambia(a[i], a[min]);
    }
}
```

Questo algoritmo effettua O(N^2) confronti ed N spostamenti, ed è quindi detto un algoritmo *quadratico*. Il fatto che sia quadratico significa che per grossi numeri di elementi l'algoritmo si comporta male, cioè impiega molto tempo. D'altra parte, questo algoritmo garantisce che al più N spostamenti verranno effettuati, che è il minimo che si può ottenere. Questo è molto importante nel caso in cui lo spostamento degli elementi sia un'operazione molto costosa rispetto al confronto, in pratica nel caso di archivi su file in cui gli elementi siano molto grossi e le chiavi molto piccole.

Questo algoritmo è adatto ad ordinare array fino a qualche decina di elementi, o qualche centinaio per programmi usa e getta. Oltre queste lunghezze è consigliabile rivolgersi ad algoritmi più sofisticati, o a routine di libreria.

[22] lunedì 26 aprile, ore 16:00-17:00, aula AI1 (1S)

Laboratorio: il filtro hexdump

Analisi ed uso del programma **`hexdump`**, coadiuvata dai programmi **`a2bin`** e **`bin2a`**:

Sorgente: a2bin.cc

```
#include <iostream>
#include <fstream>

// Leggi da standard input in ascii e scrivi su file in binario
int
main (int argc, char *argv[])
{
  if (argc != 2)
    { cerr << "Un argomento richiesto: nome del file di uscita\n"; exit (1); }
  ofstream f (argv[1], ios::bin|ios::trunc|ios::out);
  double x;
  while (cin >> x)
    f.write(&x, sizeof(x));
  return 0;
}
```

Sorgente: bin2a.cc

```
#include <iostream>
#include <fstream>

// Leggi da file in binario e scrivi su standard input in ascii
int
main (int argc, char *argv[])
{
  if (argc != 2)
    { cerr << "Un argomento richiesto: nome del file di ingresso\n"; exit (1); }
  ifstream f (argv[1], ios::bin|ios::in);
  double x;
  while (f.read(&x, sizeof(x)))
    cout << x << '\n';
  return 0;
}
```

[23] mercoledì 28 aprile, ore 14:00-16:00, aula A32 (2L)

La ricorsione

Ogni volta che una funzione viene chiamata, durante l'esecuzione di un programma in C, essa alloca dello spazio in un'area di memoria chiamata *stack*, che è appunto gestita come una pila. La funzione alloca sullo stack l'indirizzo di ritorno, cioè il punto del codice a cui deve tornare quando esegue l'istruzione `return`. Inoltre alloca spazio per gli argomenti della funzione stessa, e per tutte le variabili automatiche della funzione.

Sullo stack, quindi, sono presenti le variabili automatiche della funzione in esecuzione, quelle della sua chiamante, che tornerà in esecuzione quando la funzione corrente ritornerà, quelle della chiamante della chiamante, e così via.

Se una funzione chiama se stessa, viene detta un funzione *ricorsiva*. Ogni *istanza* della funzione alloca sullo stack l'indirizzo di ritorno, i propri argomenti, e le proprie variabili automatiche. Questo significa che le variabili automatiche sono effettivamente variabili diverse per ogni istanza della funzione.

L'esempio classico che si usa per esemplificare il concetto delle funzioni ricorsive è il calcolo ricorsivo del fattoriale. Si dice *fattoriale* di un numero *N*, e si rappresenta con la notazione matematica *N*!, il prodotto dei numeri interi da 1 ad *N*. Una formulazione ricorsiva della definizione del fattoriale è la seguente:

```
N! = N·(N-1)!, 0! = 1
```

Analogamente a tutte le formulazioni ricorsive, è composta di una definizione del fattoriale in termini del fattoriale stesso, e di una condizione terminale. Questa definizione si può facilmente implementare in C come segue:

Sorgente: fatt.cc

```
int
fattoriale (int n)
{
  if (n == 0)
    return 1;
  else
    return fattoriale(n-1);
}
```

Un esempio più concreto è un'implementazione ricorsiva della ricerca binaria, che risulta in un codice che rispecchia l'immagine mentale del critero di ricerca binaria, e cioè: confronta con l'elemento centrale. Se è uguale, hai finito, se è maggiore, applica lo stesso algoritmo alla metà superiore, altrimenti applica lo stesso algoritmo alla metà inferiore.

Sorgente: ricerca.cc

```
int
ricerca_binaria_ricorsiva (char *dato, anagrafico a[], int l, int r)
{
  if (l > r)
    return -1;
  int m = (l+r)/2;
  int cfr = strncmp(a[m].cognome, dato, 40);
  if (cfr == 0)
    return m;
  if (l == r)
    return -1;
  if (cfr > 0)
    return ricerca_binaria_ricorsiva(dato, a, l, m-1);
  else
    return ricerca_binaria_ricorsiva(dato, a, m+1, r);
}
```

Liste semplici: inserimento in testa, in coda, in ordine

C: 17.2; D: 19.2

Una *lista semplice* è un insieme di *nodi* connessi fra loro in modo unidirezionale. Per accedere ad una lista semplice è necessario un puntatore alla *testa* della lista, cioè al primo nodo della lista. La lista è una struttura dati particolarmente adatta a situazioni in cui bisogna allocare spazio per un numero di elementi sconosciuto a priori, ed è importante allocare non più dello spazio necessario.

È anche adatta a casi in cui l'inserzione di nuovi elementi e la cancellazione di elementi prsenti sono operazioni frequenti rispetto alla ricerca di un elemento presente. L'allocazione dinamica di un array è invece poco adatta al caso in cui avvengono spesso inserzioni o estrazioni di elementi, perché in tali casi bisogna rallocare l'array, e ciò comporta la copia degli elementi del vecchio array nel nuovo.

Ecco un esempio delle operazioni comuni su una lista semplice:

Sorgente: lista.cc

```
#include <stdlib.h>
#include <string.h>

struct anagrafico {
  char nome [20];
  char cognome [40];
  int annonascita;
  enum { M, F } sesso;
};

struct nodo
{
  nodo *next;
  anagrafico dati;
};

nodo *head = NULL;

nodo *
crea_nodo (anagrafico d)
{
  nodo *np = new nodo;
  np->next = NULL;
  np->dati = d;
  return np;
}

void
inserisci_in_testa (nodo *p)
{
  p->next = head;
  head = p;
}

void
aggiungi_in_coda (nodo *p)
```

```
{
  nodo *np;
  if (head == NULL)
    head = p;
  else
    for (np = head; np != NULL; np = np->next)
      if (np->next == NULL)
        np->next = p;
}

void
inserisci_in_ordine (nodo *p)
{
  nodo *np;
  if (head == NULL)
    head = p;
  else
    for (np = head; np != NULL; np = np->next)
      if (np->next == NULL
          || strncmp(np->next->dati.cognome, p->dati.cognome, 40) > 0)
        {
          p->next = np->next;
          np->next = p;
        }
}

nodo *
cerca_nodo (char *cognome)
{
  nodo *p;
  for (p = head; p != NULL; p = p->next)
    if (!strncmp(p->dati.cognome, cognome, 40))
      break;
  return p;
}

void elimina_nodo (nodo *p)
{
  for (nodo *np = head; np != NULL; np = np->next)
    if (np->next == p)
      {
        np->next = p->next;
        delete p;
        break;
      }
}
```

È anche utile avere un’operazione di estrazione di un nodo che tolga il nodo dalla lista, senza distruggerlo, e ne restituisca un puntatore. Questo è utile nel caso che si voglia inserire il nodo in un’latra lista. Un uso comune delle liste è infatti la gestione di diversi oggetti che vengono spostati da una lista all’altra durante l’elaborazione.

Se l’operazione di accodamento di un nuovo elemento è frequente e la lista è lunga, conviene usare un puntatore alla coda, cioè all’ultimo elemento, in aggiunta al puntatore alla testa. Quelle che seguono sono le stesse operazioni di prima, modificate per usare il puntatore lla coda.

Sorgente: listac.cc

```
#include <stdlib.h>
#include <string.h>

struct anagrafico {
  char nome [20];
  char cognome [40];
  int annonascita;
  enum { M, F } sesso;
};

struct nodo
{
  nodo *next;
  anagrafico dati;
};

nodo *head = NULL;
nodo *tail = NULL;

nodo *
crea_nodo (anagrafico d)
{
  nodo *np = new nodo;
  np->next = NULL;
  np->dati = d;
  return np;
}

void
inserisci_in_testa (nodo *p)
{
  p->next = head;
  if (head == NULL)
    tail = p;
  head = p;
}

void
aggiungi_in_coda (nodo *p)
{
  nodo *np;
  if (head == NULL)
    head = p;
  else
    tail->next = p;
  tail  = p;
}

void
inserisci_in_ordine (nodo *p)
{
  nodo *np;
  if (head == NULL)
    head = p;
  else
```

```
    for (np = head; np != NULL; np = np->next)
      if (np->next == NULL
          || strncmp(np->next->dati.cognome, p->dati.cognome, 40) > 0)
        {
          p->next = np->next;
          np->next = p;
        }
  if (np->next == NULL)
    tail = np;
}

nodo *
cerca_nodo (char *cognome)
{
  nodo *p;
  for (p = head; p != NULL; p = p->next)
    if (!strncmp(p->dati.cognome, cognome, 40))
      break;
  return p;
}

void elimina_nodo (nodo *p)
{
  for (nodo *np = head; np != NULL; np = np->next)
    if (np->next == p)
      {
        np->next = p->next;
        if (np->next == NULL)
          tail = np;
        delete p;
        break;
      }
}
```

Abbiamo visto le funzioni fondamentali che implementano una lista semplice, e una variazione che usa un ulteriore puntatore alla coda della lista in aggiunta al puntatore alla testa. Un'altra variazione comune consiste nell'usare un contatore, che contiene il numero dei nodi nella lista. Il contatore è utile in quei casi in cui nel programma che usa la lista è necessario contare il numero di nodi, il che senza contatore implica la completa scansione della lista. Se si usa un contatore, il test di lista vuota può esser fatto controllando se il contatore è pari a 0, invece che controllando se il puntatore alla testa è NULL.

Lista doppia

Se i nodi della lista, olre a contenere un puntatore al prossimo nodo, cioè un *puntatore in avanti*, contengono anche un puntatore al precedente elemento, cioè un *puntatore all'indietro*, si parla di *lista doppia*. Una lista doppia è utile nel caso in cui ci si trova spesso nella necessità di trovare un elemento che precede un elemento dato, perchè la lista può essere efficientemente scandita in entrambi i sensi. La gestione di una lista doppia è più complessa di quella di una lista semplice, quindi è il caso di usarla solo se effettivamente serve. Se esiste la necessità di effettuare scorrimenti all'indietro, certamente srà utile disporre di un puntatore alla coda, per cui l'implementazione di una lista doppia tipicamente prevede l'uso di un puntatore alla testa e di uno alla coda. Il puntatore in avanti del'ultimo elemento è NULL, così come il puntatore all'indietro del primo elemento.

Strutture dati astratte

Si intende come *struttura dati astratta* un modello concettuale di ordinare dei dati, che può essere implementato mediante l'uso di diverse strutture dati fra quelle che abbiamo visto finora, e cioè l'array, la struct, la lista, la lista doppia.

Ad esempio, abbiamo esaminato due diverse implementazioni della struttura dati astratta *matrice*: la prima utilizza gli array bidimensionali del C, la seconda un array di puntatori ad array. Usando due tipi di strutture dati diverse, i due metodi implementano lo stesso tipo di dati astratto.

Esaminiamo ora alcuni tipi comuni di tipi di dati astratti e alcune loro possibili implementazioni.

L'anello

Un *anello* (*ring*) è una struttura composta da elementi in sequenza lineare, ognuno dei quali ha un sucecssivo ed un precedente, tale che la sequenza sia chiusa, cioè non esista un primo ed un ultimo elemento.

Nell'esempio della media mobile abbiamo visto come implementare un anello usando un array. In un array di N elementi, ogni elemento di indice i ha un successivo, che è l'elemento di indice i+1, e un preceednte, che è l'elemento di indice i-1. Inoltre, il precedente dell'elemento di indice 0 è l'elemento di indice N-1, e viceversa.

Un anello può essere implementato in modo naturale anche con una lista, in cui il puntatore a successivo dell'ultimo elemento punti al primo. Se è necessario scorrere l'anello in entrambi i sensi, è utile utilizzare una lista doppia, in cui il puntatore al precedente del primo elemento punti all'ultimo. Un anello implementato con una lista può crescere e decrescere di dimensione in maniera semplice, aggiungendo o togliendo nodi.

La coda

Un *coda* (*queue*) è una struttura dati in cui si possono inserire e togliere elementi, eventualmente con capienza limitata. L'operazione di estrazione può essere fatta su un solo elemento, cioè su quello che è stato immesso per primo di quelli presenti nella coda. Per questa ragione si parla spesso di *coda FIFO* (First In, First Out).

La coda di macchine che si forma ad un semaforo su una sola corsia ne è un esempio: Una sola macchina alla volta può essere inserita in coda, e una sola alla volta ne può uscire. Quella che ne può uscire è quella che è in coda da più tempo.

La maniera più intuitiva per implementare una coda utilizza una lista semplice, con inserzione in coda ed estrazione dalla testa (fare al contrario è più complicato). Una coda sifatta non ha dimensione massima, se non quella massima della memoria.

Un modo più efficiente di implementare una coda è di usare un array con due indici, uno relativo all'elemento più vecchio in coda ed uno relativo al primo posto libero, insieme con un contatore degli elementi in coda. Il primo indice avanza ogni volta che si estra un elemento dalla coda, il secondo ogni volta che ne si aggiunge uno. Quando un indice arriva all'ultimo elemento, per avanzare lo si riporta al

primo. Una coda così implementata occupa sempre lo stesso spazio in memoria, indipendentemente da numero degli elementi in coda, ed ha una capienza massima pari alla dimensione dell'array. Il contatore serve a distinguere i casi di coda piena e coda vuota, perchè in entrambi i casi i due indici coincidono.

La pila

Un *pila* (*stack*) è una struttura dati in cui si possono inserire e togliere elementi, eventualmente con capienza limitata. L'operazione di estrazione può essere fatta su un solo elemento, cioè su quello che è stato immesso per ultimo di quelli presenti nella coda. Per questa ragione si parla spesso di *coda LIFO* (Last In, First Out).

I distributori di confezioni di caramelle nei negozi sono un esempio di pila: se vengono caricati dalla stessa parte da cui le confezioni sono prelevate dal cliente, quella che ogni volta si può prendere è quella che è in coda da meno tempo.

Un modo di implementare una pila utilizza una lista semplice, con inserzione ed estrazione dalla testa. Una pila sifatta non ha dimensione massima, se non quella massima della memoria.

Un modo più efficiente di implementare una pila è di usare un array, con un indice relativo al primo posto libero. L'indice avanza ogni volta che si aggiunge un elemento alla pila, e retrocede quando si toglie un elemento. La pila è vuota quando l'indice è pari a 0, è piena quando l'indice è pari alla dimensione dell'array.

[24] lunedì 3 maggio, ore 9:30-11:30, aula A23 (2L)

Il foglio elettronico come matrice di celle

C: 20.1

Il foglio elettronico è una matrice di celle. I prodoti attuali tipicamente consentono di creare un documento con più fogli. Ogni cella è quindi individuata dalle coordinate di foglio, riga, colonna. Se il foflio si chiama «pot», la coordinata della (d'ora in poi *riferimento alla*) prima cella sarà `pot!R1C1`. Se il nome del foglio ed il punto esclamativo sono omessi, si intende che il riferimento è al foglio corrente. D'ora in poi ometteremo sempre il riferimento al foglio.

Riferimenti assoluti e relativi

I riferimenti possono essere assoluti, relativi o misti. Un *riferimento assouto*, del tipo `R4C23` si riferisce alla cella in riga 4, colonna 23. Un *riferimento relativo*, del tipo `RC[-3]`, si riferisce alla cella nella stessa riga del riferimento e 3 colonne sopra. I numeri negativi si riferiscono a righe superiori o a colonne alla sinistra. Un *riferimento misto*, del tipo `R5C[12]`, si riferisce alla cella in riga 5, 12 colonne sotto. Ci sono due tipi di riferimento misto, uno con riga assoluta ed uno con colonna assoluta.

Quando una formula viene copiata in più celle essa rimane uguale, per cui diventa importante distinguere fra riferimenti assoluti e relativi. Per default, i fogli elettronici usano i riferimenti relativi, che sono di uso più comune.

Numeri, etichette, formule; contenuto e valore

C: 20.1.1, 20.2

Il contenuto di una cella può essere di diversi tipi, fra cui un *numero*, una *stringa*, una *formula*. Le stringhe sono anche dette *etichette*. I numeri possono esere rappresentati in diversi formati, fra cui il formato *percentuale* e il formato *data*. Le date infatti sono rappresentati come numero di giorni a partire da una data di riferimento, tipicamente il 1º gennaio 1900.

Quando il contenuto di una cella è una formula, normalmente invece del *contenuto* viene visualizzato il *valore*, cioè il risultato della formula. Esiste una gran varietà di formule, per usi matematici, ingegneristici, finanziari, statistici, di gestione di tabelle di archivi.

Intervalli, formule con intervalli

È possibile specificare un'area rettangolare del foglio, detto *intervallo* (*range*), indicandone la cella in alto a sinistra e quella in basso a destra, così: `R2C4:R9C5`. L'esempio è il riferimento ad un'area rettangolare `8x2`, cioé di 8 righe e due colonne. In Excel, è anche possibile creare intervali formati da più zone rettangolari disgiunte, così: `(R2C4:R9C5,R4C9:R6C27)`. Gli intervalli sono necessari per le funzioni che operano su un numero variabile di valori di input. Ad esempio, la funzione `SOMMA` prende un numero arbitrario di argomenti, ognuno dei quali è una cella, un numero, o un intervallo: `=SOMMA(2;R1C2;R2C4:R9C5)`. Il valore dei questa formula è pari a 2 più il valore della cella R1C2 più la somma dei valori delle celle nell'area R2C4:R9C5.

Etichette come nomi di variabile

È possibile assegnare un nome simbolico ad un riferimento, sia esso assoluto, relativo, o misto. Una volta ssegnato un nome, esso può essere usato nelle formule, rendendole più leggibili. Per fogli elettronici non semplicissimi o non del tipo usa-e-getta è sempre consigliabile usare nomi simbolici quando possibile.

Ricerca obiettivo

Esiste in Excel una funzionalità detta (a seconda delle versioni) *ricerca obiettivo* o *risolutore*. SI tratta di un algoritmo di ricerca degli zeri piuttosto sofisticato, il cui uso però è spesso semplice ed intuitivo. In generale, se una cella contiene una formula dipendente, direttamente o indirettamente, da un numero posto in un'altra cella, per mezzo di una o più formule, è possibile variare il contenuto della cella indipendente in modo da ottenere il valore desiderato nella cella dipendente (quella con la formula). Sono quindi necessari tre parametri per il risolutore: il valore che si vuole ottenere nella cella dipendente, il riferimento alla cella dove si vuole ottenere quel valore (la cella dipendente), ed il riferimento alla cella che bisogna cambiare per ottenere quel valore (la cella indipendente).

Esempio: media mobile

Data una sequenza di numeri in una colonna del foglio, calcolare nella colonna alla sua destra la media mobile dei numeri con una finestra di lunghezza 5.

Nelle celle alla destra della sequenza di numeri, a partire dalla quinta riga, si dovrà immettere la formula `=MEDIA(R[-4]C[-1]:RC[-1])` . Dentro le celle sono indicati i contenuti, non i valori:

	53.8	
	-56	
	52	
	84.47	
	22	=MEDIA(R[-4]C[-1]:RC[-1])
	62.4	=MEDIA(R[-4]C[-1]:RC[-1])
	88	=MEDIA(R[-4]C[-1]:RC[-1])

Esempio: filtro FIR

Data una sequenza di numeri in una colonna del foglio, calcolare nella colonna alla sua destra l'uscita del filtro FIR del secondo ordine i cui tre coefficienti sono contenuti nela regione `R1C1:R3C1`.

Nelle celle alla destra della sequenza di numeri, a partire dalla terza riga, si dovrà immettere la formula `=R[-2]C[-1]*R1C1 + R[-1]C[-1]*R2C1 + RC[-1]*R3C1` . Dentro le celle sono indicati i contenuti, non i valori:

	1			
1	1.745			
2	-0.034			
3	0.452			
			52	
			84.47	
			22	=R1C1*R[-2]C[-1]+R2C1*R[-1]C[-1]+R3C1*RC[-1]
			-0.32	=R1C1*R[-2]C[-1]+R2C1*R[-1]C[-1]+R3C1*RC[-1]
			84	=R1C1*R[-2]C[-1]+R2C1*R[-1]C[-1]+R3C1*RC[-1]

Esempio: calcolo del capitale maturato

Supponiamo di utilizzare l'intervallo 101 righe e 11 colonne `R1C1:R101C11` per calcolare il valore di un deposito nel tempo. Il tempo sia espresso in numero di anni (valori *x*) nell'intervallo `R1C2:R1C11`, ed i diversi importi iniziali del capitale siano i valori *y* nell'intervallo `R2C1:R101C1`. Il calcolo è effettuato per un dato tasso di interesse, contenuto nella cella `R1C1`.

Precisamente, in tutte le celle dell'intervallo `R2C2:R101C11` sarà contenuta la formula `=RC1*(1+R1C1)^R1C` . Dentro le celle sono indicati i contenuti, non i valori:

	1	**2**	**3**		**10**	**11**	**12**
1	3%	1	2		9	10	
2	100	=RC1*(1+R1C1)^R1C	=RC1*(1+R1C1)^R1C		=RC1*(1+R1C1)^R1C	=RC1*(1+R1C1)^R1C	
3	200	=RC1*(1+R1C1)^R1C	=RC1*(1+R1C1)^R1C		=RC1*(1+R1C1)^R1C	=RC1*(1+R1C1)^R1C	
100	9900	=RC1*(1+R1C1)^R1C	=RC1*(1+R1C1)^R1C		=RC1*(1+R1C1)^R1C	=RC1*(1+R1C1)^R1C	
101	10000	=RC1*(1+R1C1)^R1C	=RC1*(1+R1C1)^R1C		=RC1*(1+R1C1)^R1C	=RC1*(1+R1C1)^R1C	
102							

Si noti che la formula contiene un riferimento assoluto e due misti, uno con riga assoluta e l'altro con colonna assoluta.

Una volta costruita un tale tabella, con ricerca obiettivo si può trovare il tasso di interesse tale che ad una certa data, con un certo capitale iniziale, si ottenga un dato capitale finale. Per esempio, vogliamo trovare il tasso di interesse tale che, dopo 9 anni, un capitale iniziale di 7500€ sia diventato pari a 10000€. Il valore che vogliamo ottenere è 10000, la cella dipendenteo (che contiene una formula) è la cella `R76C10`, e la cella indipendente (che contiene un numero) è la cella `R1C1`. Alla fine del processo di soluzione, nella cella `R1C1` troveremo il valore 3.25%.

Funzioni di gestione del foglio elettronico

Sono possibili numerosi operazioni, fra cui spostamento, inserimento e cancellazione di celle, range, righe e colonne. È inoltre generalmente facile produrre grafici e diagrammi di serie di dati in vari formati, come diagrammi a barre orizzontali e verticali, a torta, diagrammi XY. Il tipo di grafico di maggiore utilità per applicazioni numeriche è il grafico XY, che richiede una (o più) serie di valori X e una (o più) serie corrispondenti di valori Y.

[25] lunedì 3 maggio, ore 16:00-17:00, aula AI1 (1S)

Laboratorio: esempi con Excel

[26] mercoledì 5 maggio, ore 14:00-16:00, aula A32 (1L,1E)

Esercizio: numeri di Fibonacci

Calcolare la sequenza così definita:

$$x_1 = 0$$
$$x_2 = 1$$
$$x_n = x_{n-1} + x_{n-2}$$

facendo in modo che la formula generatrice possa essere copiata in un arbitrario numero di celle, cioè che sia sempre la stessa.

Ecco una soluzione. Dentro le celle sono indicati i contenuti, non i valori:

	0
	1
	=R[-1]C+R[-2]C
	=R[-1]C+R[-2]C
	=R[-1]C+R[-2]C

Esercizio: numeri di Neper

Calcolare la sequenza così definita:

$$e_1 = 2$$
$$e_n = e_1 \cdot e_2 \cdot \ldots \cdot e_{n-1} + 1$$

facendo in modo che la formula generatrice possa essere copiata in un arbitrario numero di celle, cioè che sia sempre la stessa.

Ecco una soluzione. Dentro le celle sono indicati i contenuti, non i valori:

1	2	
2	=PRODOTTO(R1C:R[-1]C)+1	
3	=PRODOTTO(R1C:R[-1]C)+1	
4	=PRODOTTO(R1C:R[-1]C)+1	
5		

Excel: stile tradizionale di riferimento

Esistono in Excel due stili di riferimento. Quello che abbiamo visto è detto stile «`R1C1`», l'altro è lo stile «`A1`». In questo secondo stile, che è quello tradizionale, ed usato per default, i numeri di colonna sono sostituiti da lettere: `A`, `B`, …, `Z`, `AA`, `AB`, …, `AZ`, `BA`, e così via. A parte la nomenclatura, questo stile utilizza un metodo diverso per distinguere fra riferimenti assoluti e relativi. Infatti, un riferimento assoluto alla cela in riga 3, colonna 14 è scritto come `$N$3` (`N` è la colonna 14). Il simbolo `$` sta ad indicare che il riferimento è assoluto. Senza il simbolo `$` i riferimenti sono invece relativi. Quindi, se nella cella `A2` troviamo un riferimento ad `A1`, questo significa «riferimento alla cella sopra». Se la formula contenente il riferimento `A1` viene copiata nelle celle sottostanti, essa cambia, predendo ogni volta il nome della cella cui si riferisce. Per esempio, nella cella `A14` il riferimento copiato diventerà `A13`.

Ecco come produrre i numeri di Fibonacci usando lo stile di riferimento `A1`:

	A	
1	0	
2	1	
3	=A1+A2	
4	=A2+A3	
5	=A3+A4	
6		

Ed ecco come produrre i numeri di Neper usando lo stile di riferimento `A1`:

	A
1	2
2	=PRODOTTO(A$1:A1)+1
3	=PRODOTTO(A$1:A2)+1
4	=PRODOTTO(A$1:A3)+1
5	

In entrambi i casi, la formula generatrice può essere copiata in un arbitrario numero di celle, cioè è sempre la stessa, anche se apparentemente cambia di cella in cella.

Si noti come in entrambi gli esempi non si possano scrivere le formule prescindendo dalla posizione nel foglio elettronico, mentre negli esempi con lo stile `R1C1` le formule dei numeri di Fibonacci potevano prescinderne completamente (formula con riferimenti tutti relativi) e la formula di Neper poteva prescinderne per quanto riguardava le colonne (formula con riferimenti di colonna relativi).

Formule matrice

Esistono funzioni predefinite che restituiscono più di un valore, che non consideriamo oltre, e si possono creare formule che restuiscono più di un valore, dette *formule matrice*, che invece esaminiamo. Le formule matrice di Excel sono come le formule normali, ma sono racchiuse fra parentesi graffe e al posto dei riferimenti a cella ci possono essere dei riferimenti a intervalli.

Esempio di formula matrice ad una dimensione: parabola

Nelle formule matrice ad una dimensione, i riferimenti ad intervallo sono tutti della stessa forma, ed il risultato della formula matrice copre un intervallo ancora della stessa forma. Vogliamo costruire punto per punto una parabola, ammettendo che i tre coefficienti della parabola si trovino nell'intervallo `R1C1:R1C3`. Supponiamo quindi di occupare l'intervallo `R3C4:R103C4` con i valori *x* della parabola, e di voler ottenere i valori *y* nella colonna accanto, usando una formula matrice:

	1	2	3		
1	0.49	4.7	-1.2		
2					
3				-5.0	{=R1C1*R3C4:R103C4^2+R1C4*R3C4:R103C4+R1C3}
4				-4.9	{=R1C1*R3C4:R103C4^2+R1C4*R3C4:R103C4+R1C3}
102				4.9	{=R1C1*R3C4:R103C4^2+R1C4*R3C4:R103C4+R1C3}
103				5.0	{=R1C1*R3C4:R103C4^2+R1C4*R3C4:R103C4+R1C3}
104					

Si noti che la formula usa due riferimenti intervallo, entrambi della stessa forma, che è anche la stessa forma del valore della formula. Come in tutte le formule matrice, il valore della formula è un intervallo di valori, per cui la formula deve occupare più celle per visualizzare il proprio valore. I riferimenti intervallo sono tutti assoluti. Questa scelta è fatta per semplificare la formula, ma non c'è alcun vincolo sul tipo di riferimenti presenti in una formula matrice, che possono essere di qualunque tipo, assoluto, relativo o misto, e possono essere di tipi diversi fra loro. Se si usano dei riferimenti relativi per indicare gli intervali, questi sono relativi alla prima cella (in alto a sinistra) dell'intervallo occupato dalla formula matrice.

Esempio di formula matrice a due dimensioni: calcolo del valore maturato

Se in una formula matrice esistono riferimenti a intervalli di due (e solo due) forme diverse, e precisamente intervalli di dimensione `1xM` e intervalli di dimensione `Nx1`, il valore della formula matrice sarà ottenuto componendo i valori contenuti in questi intervalli con tutte le combinazioni possibili, e sarà un intervallo di dimensione `NxM`. In altre parole, nella formla devono esistere riferimenti a intervalli di dimensione una riga per *M* colonne e riferimenti a intervalli di *N* righe per una colonna. Il risultato sarà un intervallo rettangolare di *N* righe per *M* colonne.

Il calcolo del valore di un deposito per diversi capitali iniziali e diversi numeri di anni di deposito, ad un dato tasso di interesse, può essere riscritto con una singola formula matrice che occupa l'intero intervallo `R2C2:R101C11`:

	1	**2**		**11**	**12**
1	3%	2		10	
2	100	{=R2C1:R101C1*(1+R1C1)^R1C2:R1C11}		{=R2C1:R101C1*(1+R1C1)^R1C2:R1C11}	
3	200	{=R2C1:R101C1*(1+R1C1)^R1C2:R1C11}		{=R2C1:R101C1*(1+R1C1)^R1C2:R1C11}	
100	9900	{=R2C1:R101C1*(1+R1C1)^R1C2:R1C11}		{=R2C1:R101C1*(1+R1C1)^R1C2:R1C11}	
101	10000	{=R2C1:R101C1*(1+R1C1)^R1C2:R1C11}		{=R2C1:R101C1*(1+R1C1)^R1C2:R1C11}	
102					

[27] lunedì 10 maggio, ore 9:30-11:30, aula A23 (2L)

I database: tabelle, campi, record

Una *tabella* è un insieme di informazioni così organizzato:

Libri		
Titolo	**Autore**	**Editore**
Excel 97	Jacobson	Microsoft
The C++ programming language	Stroustroup	Addison-Wesley
Informatica istituzioni	Ceri	McGraw-Hill
Introduzione alla programmazione	Domenici	Angeli

Quella rappresentata è la tabella «Libri», con tre *campi*, che sono «Titolo», «Autore», «Editore», e quattro *record*, ognuno dei quali descrive un libro.

La *struttura di una tabella*, nella terminologia di Access, è l'elenco dei suoi campi. I *dati di una tabella* sono l'insieme dei record che costituiscono quella tabella. Una *struttura di database*, nella terminologia di Access, è l'insieme delle strutture delle tabelle presenti nel database e delle loro connessioni. Ecco una rappresentazione della struttura della tabella «Libri»:

Libri
Titolo
Autore
Editore

Per evitare duplicazione dell'informazione, normalmente conviene usare tabelle diversi per cose diverse. Per esempio, un database dei libri di una biblioteca comprendente titolo, autore e data di nascita, data di stampa, editore e suo indirizzo, posizione negli scaffali, sarebbe naturalmente suddiviso in una tabella di libri, una di autori, ed una di editori. In tal modo si evita di ripetere, per ogni libro di una carta casa editrice, l'indirizzo di quell'editore.

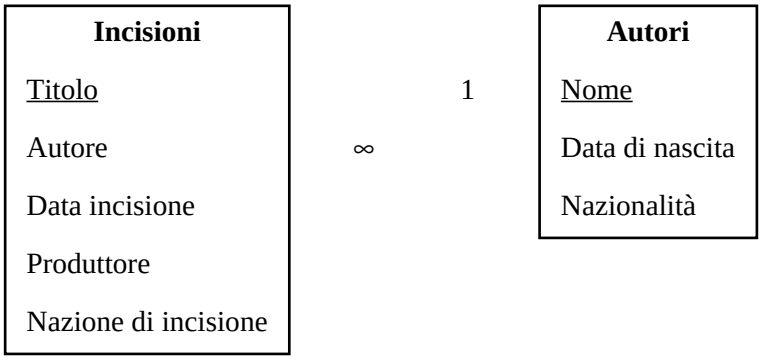
Chiavi, chiave primaria, relazioni, chiave esterna

In una tabella, i record sono unici, cioè non ne esistono due uguali. Una chiave è un insieme minimo di campi che identifica univocamente un record. Fra le possibili chiavi, se ne sceglie una, detta *chiave primaria*, che viene normalmente usata per definire le *relazioni* fra le tabelle. Nella rappresentazione della struttura della tabella «Libri», la chiave primaria «Titolo» è sottolineata.

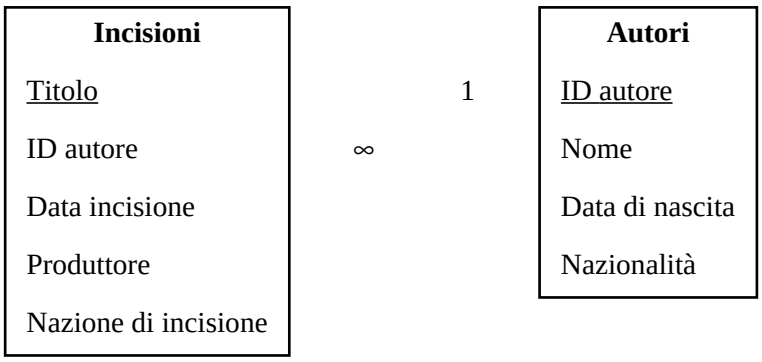
Quando una tabella A ha una relazione con un'altra tabella B, in entrambe esiste un campo, normalmente con lo stesso nome in entrambe le tabelle. Questo campo è normalmente un chiave primaria per una delle due, ad esempio la tabella A, e non è una chiave per l'altra. Questo campo nella tabella B è detto *chiave straniera*.

Relazioni uno a molti

Supponiamo di costruire un database di dischi e dei loro autori. Ogni disco ha un autore, ma siccome ci sono più dischi con lo stesso autore, usiamo due tabelle per evitare eccessiva duplicazione di informazione. Le due tabelle sono «Incisioni» e «Autori», ed hanno la seguente struttura:



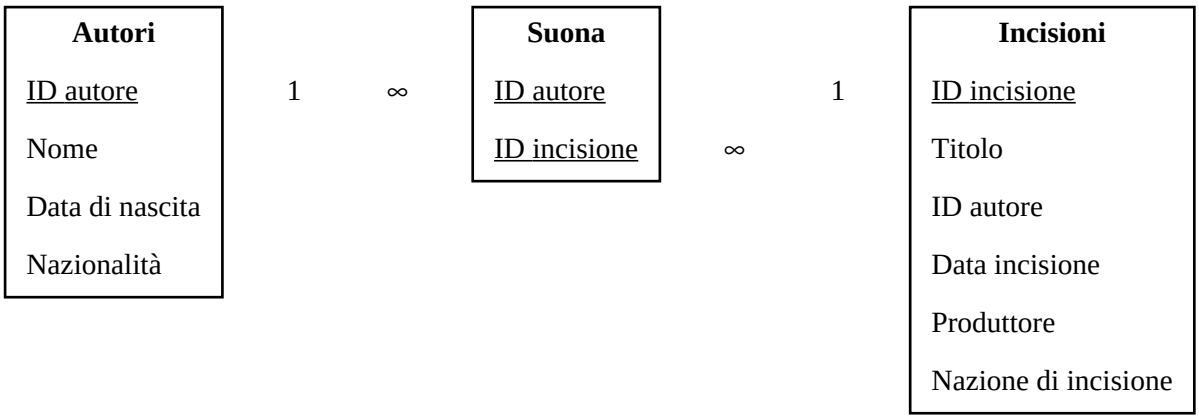
In questo esempio, i campi `Incisioni.Autore` e `Autori.Nome` sono legati da una relazione *uno a molti*, cioè ad ogni record della tabella «Autori» (lato uno della relazione) può essere connesso più di un record della tabella «Incisioni» (lato molti della relazione). La relazione avviene fra un campo dalla parte uno, che sia una chiave primaria, e un campo dalla parte molti che è quindi una chiave esterna. Si evita duplicazione dell'informazione perché tutte le informazioni relative alla data di nascita e alla nazionalità di un autore sono presenti una sola volta per ogni autore. Si può fare di meglio usando un identificativo per gli autori, cioè un numero o un breve codice che identifichi univocamente un autore:



In questo caso, la relazione è fra i campi `Incisioni.ID autore` (chiave primaria, dalla parte uno) e `Autori.ID Autore` (chiave esterna, dalla parte molti).

Relazioni molti a molti, tabella ponte

Se vogliamo considerare il caso di dischi con più autori, fra le tabelle «Autori» e «Incisioni» esiste una relazione *molti a molti*. Siccome fra tabelle non possono esistere direttamente relazioni molti a molti, bisogna costruire una *tabella ponte*, che consenta di costruire la relazione molti a molti usando due relazioni uno a molti. La tabella ponte indica il modo con cui le due tabelle «Incisioni» e «Autori» sono correlate. A tal fine, aggiungiamo un identificatore anche alla tabella «Incisioni»:



La tabella «Suona» ha una chiave primaria composta di due campi, «ID autore» e «ID incisione», perché nessuno dei due campi, preso da solo, identifica univocamente un record. Questa è la situazione normale in una tabella ponte.

Esistono due relazioni: la prima è fra `Autori.ID autore` (chiave primaria, dalla parte uno) e `Suona.ID autore` (chiave esterna, dalla parte molti). La seconda è fra `Incisioni.ID incisione` (chiave primaria, dalla parte uno) e `Suona.ID incisione` (chiave esterna, dalla parte molti).

Regole di integrità

Usando le informazioni presenti nella struttura, un programma di gestione di database può essere in grado di verificare alcune regole che consentono di mantenere la coerenza delle informazioni presenti nel database, ogni volta che si aggiungono dati, cioè che si riempiono dei record in una tabella.

Un primo controllo avviene se si è definita una chiave primaria, e consiste nel verificarne *l'unicità*. Ogni volta che si aggiunge un record, il programma verifica che la sua chiave abbia un valore diverso da quello di tutti i record già presenti nella tabella.

Un altro controllo, detto di *integrità relazionale*, avviene se è presente una relazione uno a molti. Ogni volta che si aggiunge un record in una tabella che sta dalla parte molti di una relazione, si verifica che la chiave esterna corrisponda ad una chiave primaria nella tabella che sta dalla parte uno, cioè che esista un record con quell'identificatore.

Query

Lo scopo della costruzione di un database è di poterlo *interrogare*, cioè di poterne estrarre delle informazioni selezionate. Un esempio di interrogazione nel nostro caso sarebbe di chiedere quali sono i titoli e le date di incisione dei dischi incisi dopo una certa data. Il risultato è fornito ancora in forma tabellare, con una riga (un record) per ogni incisione che soddisfa le condizioni. Ogni record del risultato della interrogazione sarà composto di due campi (i campi), cioè il titolo e la data di incisione.

Una semplice *query* in Access è costituita concettualmente di tre fasi effettuate in sequenza: *join*, *selezione*, *proiezione*. Consideriamo prima una semplice query che richieda solo l'ultima fase.

Proiezione

L'operazione di *proiezione* di una tabella secondo uno o più campi consiste nel produrre una nuova tabella costituita solo di quei campi, i cui record siano ancora unici. In Access viene semplicemente chiamata visualizzazione, ma visualizza anche valori ripetuti. Perché Access effettui una vera operazione di proiezione, bisogna che nella riga «formula:» della struttura della query sia selezionato «Raggruppamento». Si tratta di un particolare del metodo usato da Access, che nel seguito trascuriamo.

Supponiamo quindi di volere l'elenco delle nazioni di incisione dei dischi posseduti. Sarà sufficiente eseguire una proiezione della tabella «Incisioni» secondo il campo «nazione di incisione». Il risultato sarà una tabella con un solo campo, i cui record saranno unici, cioè se ci sono più dischi incisi in una data nazione, quella nazione sarà presente una sola volta nel risultato della proiezione.

Questa è la struttura dell'interrogazione descritta, in cui il risultato viene presentato in ordine alfabetico:

Incisioni
ID incisione
Titolo
ID autore
Data incisione
Produttore
Nazione di incisione

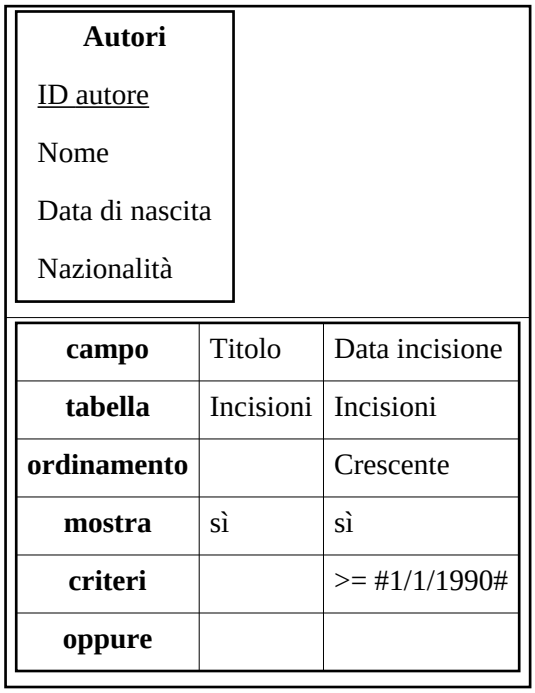
campo	Nazione di incisione
tabella	Incisioni
ordinamento	Crescente
mostra	sì
criteri	
oppure	

Selezione

L'operazione di selezione da una tabella secondo uno o più *criteri di selezione* consiste nel produrre una nuova tabella costituita dai soli record che soddisfano i criteri. I criteri possono essere i più vari, e sono delle operazioni logiche che danno risultato vero o falso.

Supponiamo quindi di volere l'elenco in ordine di data dei titoli dei dischi e della corrispondente data di incisione per tutti i dischi incisi dopo il 1º gennaio 1990. Per ottenere questo risultato bisognerà prima effettuare una selezione sulla tabella «Incisioni», con criterio `Incisioni.Data incisone >= #1/1/1990#`. Questa operazione genera, internamente al programma, una tabella con tutti i campo della tabella «Incisioni», ma contenente solo i record che soddisfano il criterio di selezione. Di questa tabella bisognerà quindi effettuare una proiezione sui campi «Titolo» e «Data incisione», ordinando secondo quest'ultimo campo.

Le due operazioni (selezione e proiezione) costituiscono l'interrogazione, e la struttura dell'interrogazione è descritta in maniera sintetica in maniera analoga alla precedente:



campo	Titolo	Data incisione
tabella	Incisioni	Incisioni
ordinamento		Crescente
mostra	sì	sì
criteri		>= #1/1/1990#
oppure		

Rispetto all'interrogazione precedente, qui ci sono due colonne di dati nella struttura, perchè si chiede di visualizzare due campi, invece di uno. Inoltre, è presente un'espressione in uno campo di selezione. L'operatore di selezione `>=` è un operatore binario, il cui primo argomento è il campo della colonna in cui si trova, in questo caso `Autori.Data incisione`.

Se più criteri di selezione si trovano nella stessa riga, in colonne successive, essi vengono composti usando l'operatore logico `AND`. I criteri di selezione che si trovano su righe diverse vengono composti usando l'operatore logico `OR`. Ad esempio, se voglio modificare il criterio di selezione della query precedente aggiungendo tutti i dischi incisi in Italia, avrò la seguente struttura della query:

Autori
ID autore
Nome
Data di nascita
Nazionalità

campo	Titolo	Data incisione	Nazionalità
tabella	Incisioni	Incisioni	Incisioni
ordinamento		Crescente	
mostra	sì	sì	
criteri		>= #1/1/1990#	
oppure			= "Italia"

In tal modo verranno visualizzati i dischi che sono stati incisi in questo decennio o, indipendentemente dalla data, che sono stati incisi in Italia.

[28] lunedì 10 maggio, ore 16:00-17:00, aula AI1 (1S)

Laboratorio: esempi con Access

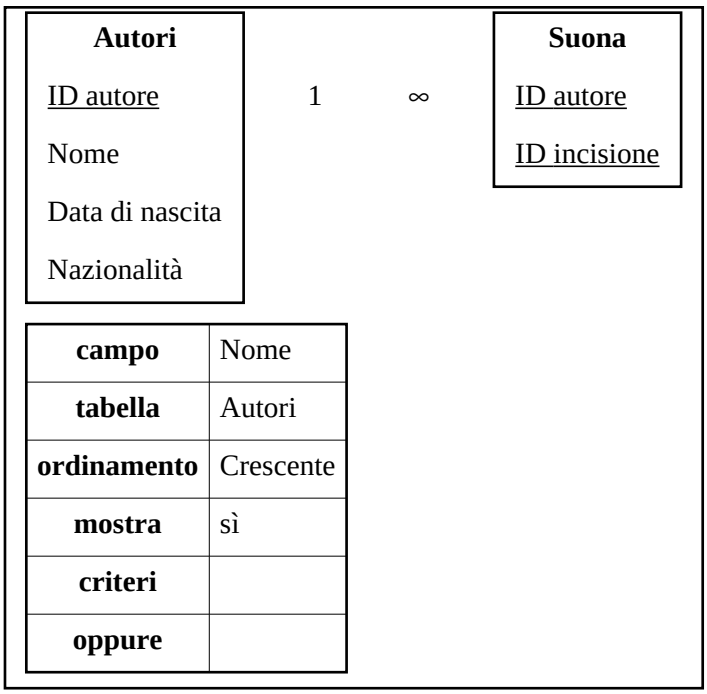
[29] mercoledì 12 maggio, ore 14:00-16:00, aula A32 (1L,1E)

Operazione di join

L'operazione di *join* fra due tabelle in relazione uno a molti fra loro costruisce una nuova tabella comprendente tutti i campi della prima e tutti i campi della seconda. I record della nuova tabella saranno tutte le possibili combinazioni dei contenuti delle due tabelle di partenza, tali che i valori dei campi in relazione siano uguali fra loro.

Quando si costruisce la struttura di un query in Access, nella metà superiore della finestra di struttura della query si visualizzano le tabelle che sono coinvolte nella query, con le opportune relazioni. Access usa le relazioni indicate per effettuare le operazioni di join, ottenendo così una tabella di join (internamente). Su questa tabella effettua quindi le operazioni già viste, cioè le selezioni e le proiezioni.

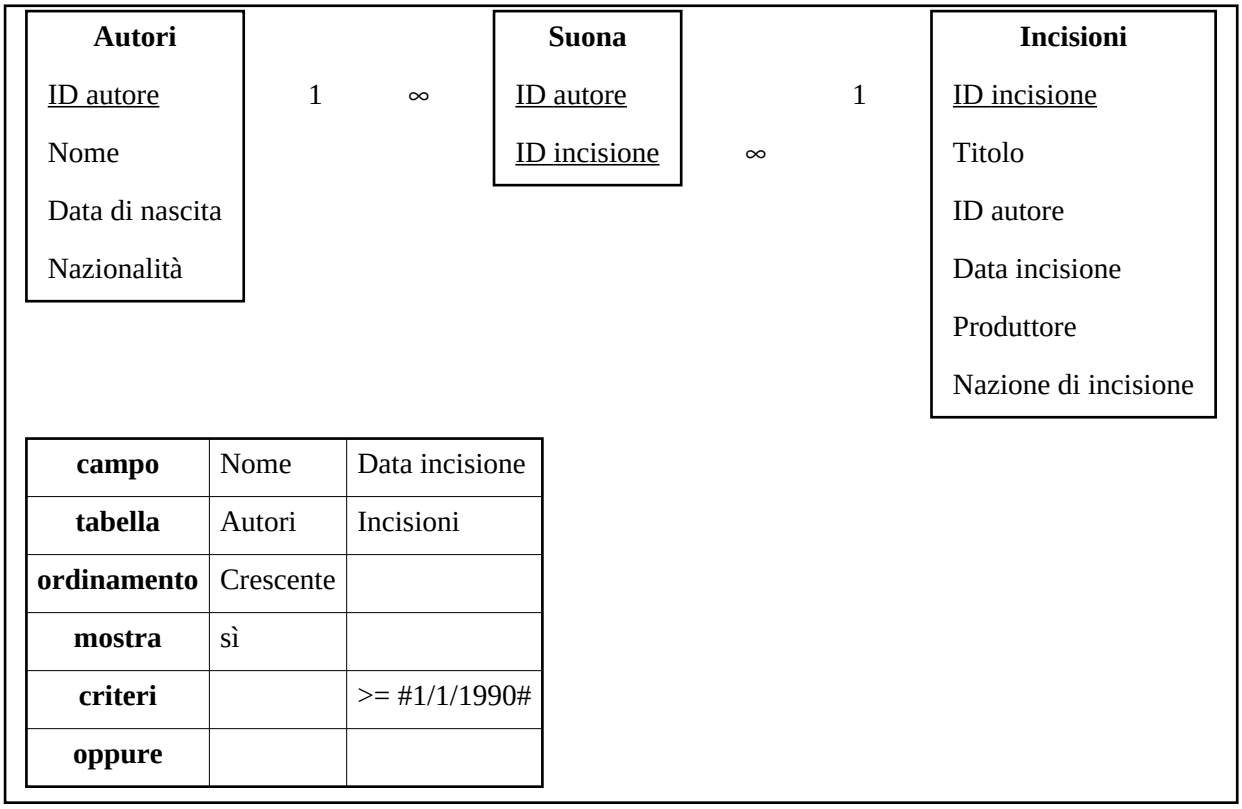
Costruiamo una query che abbia come risultato i nomi di tutti gli autori che nel database siano connessi ad almeno un'incisione.



È sufficiente specificare che si vogliono visualizzare i nomi degli autori. Il criterio di join impostato nella metà superiore della finestra, infatti, esclude tutti i record della tabella «Autori» che non abbiano almeno un corrispondente record nella tabella «Suona».

Operazione di join multipla

È anche possibile, e di uso molto comune, impostare più operazioni di join nella stessa query. Se ad esempio vogliamo sapere i nomi degli autori che ahanno effettuato incisioni nell'ultimo decennio, il programma dovrà effettuare due operazioni di join, per mettere in relazione gli autori con le corripondenti incisioni. La query risultante sarà la seguente:



campo	Nome	Data incisione
tabella	Autori	Incisioni
ordinamento	Crescente	
mostra	sì	
criteri		>= #1/1/1990#
oppure		

Esercizi su carta con Access

[30] lunedì 17 maggio, ore 9:30-11:30, aula A23 (1E,1L)

Join fra più di due tabelle

Uso di una struttura di database più complessa: