22. IO non formattato

Andrea Marongiu
(andrea.marongiu@unimore.it)
Paolo Valente

UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

I/O formattato

- Gli operatori di ingresso/uscita visti finora, ossia >> e <<, interpretano il contenuto di uno *stream* come una sequenza di caratteri
- In particolare, traducono valori in sequenze di caratteri e viceversa
- Le operazioni di ingresso/uscita in cui uno *stream* è visto come una sequenza di caratteri si definiscono operazioni di ingresso/uscita **formattate**

I/O non formattato

- Esistono anche operazioni di ingresso/uscita **non formattate**
- Vedono lo stream dal punto di vista di più basso livello
  - ossia come una mera sequenza di byte
- Non effettuano alcuna trasformazione di alcun tipo
  - Trasferiscono semplicemente sequenze di byte da uno *stream* alla memoria del processo, o dalla memoria del processo ad uno *stream*

Accesso sequenziale

- Così come le letture/scritture formattate, anche quelle non formattate accedono agli stream in modo sequenziale
  - Ogni lettura/scrittura viene effettuata a partire dal byte dello stream successivo all'ultimo byte dello stream su cui ha lavorato la lettura/scrittura precedente

Funzioni membro

- Le funzioni di lettura e scrittura non formattate che vedremo sono delle cosiddette **funzioni membro** degli stream
- Per invocarle bisogna utilizzare la notazione a punto
  - Se **f** è un [*i/o*]*stream*, allora per invocare ad esempio una funzione membro
  - **`fun(char &)`**, bisogna scrivere **`f.fun(c) ;`**
  - Esempio:
  - **`char c ;`**
  - **`cin.fun(c) ;`**

Buffer ed I/O non formattato

- Come già visto, un **buffer** è un array di byte utilizzato nelle operazioni di I/O
- Il tipo **`char`** ha esattamente la dimensione di un byte
- Per questo motivo il tipo **`char`** è utilizzato anche per memorizzare byte nelle letture e scritture non formattate

Input non formattato

- Gli istream dispongono delle seguenti funzioni membro per input non formattato:

```
get()
get(char &)
get(char *buffer, int n,
[char delimitatore])
read(char *buffer, int n)
gcount()
```

get() 1/2

- Ritorna, su un **`int`**, il valore del prossimo byte nello stream di ingresso a cui è applicata
  - Ritorna il valore **`EOF`** in caso di fine input
  - (**`EOF`** è una costante predefinita)
- Esempio con **`cin`**:

```
main()
{
  int i = cin.get() ;
  ...
}
```

get() 2/2

- Un ciclo di lettura che va avanti finché non si incontra l'EOF si può quindi scrivere:

```
main()
{
  int i ;
 while ( (i = cin.get()) != EOF )
       ...
}
```

get(char &c) 1/2

- Preleva un byte e lo assegna alla variabile passata come argomento
  - La variabile è lasciata inalterata in caso di fine input
- Esempio con **cin**:

```
main()
{
      char c ;
      cin.get(c) ;
  ...
}
```

get(char &c) 2/2

- Per semplicità, diciamo che tale funzione ritorna lo *istream* a cui è applicata
- Quindi si può usare allo stesso modo di uno *istream* in espressioni che si aspettano un booleano
- Esempio con **cin**:

```
main()
{
    char c ;
    while(cin.get(c))
        ...
}
```

Esercizio

- Dalla decima esercitazione:
  - *file/file_conta_linee.cc*

Lettura in un buffer 1/2

```
get(char *buffer, int n,
    [char delimitatore])
```

- Legge una sequenza di byte dallo stream di ingresso e li copia nell'array *buffer*, aggiungendo automaticamente il carattere `'\0'` finale
- La lettura va avanti finché non si sono letti *n* byte oppure non è stato incontrato il delimitatore
  - Se il terzo argomento non è passato, si usa come delimitatore il codice del carattere `'\n'`

Domanda

- Cosa legge quindi la funzione se non si modifica il delimitatore?

Lettura in un buffer 2/2

- Una riga di al più *n* caratteri (byte)
- Se la riga è vuota, o se in generale il numero di byte letti è zero
  - Manda l'istream in stato di errore

Esempio di lettura in un buffer

```
main()
{
     char buf[100] ;

  // lettura di al più 50 byte, a meno
  // non si incontri il codice del
  // carattere '-'
  // in fondo è inserito '\0'
     cin.get(buf, 50, '-') ;
  ...
}
```

Esempio evoluto lett. riga 1/2

- Documentandovi un po' da soli, scoprireste che
  - Mediante il precedente uso della funzione get
  - Ed utilizzando la funzione membro peek() che ritorna il valore del prossimo byte presente su un istream
    - Senza rimuovere tale byte
- Si può leggere da un *istream* una riga alla volta
  - eliminando opportunamente il carattere *newline* dopo aver consumato la riga

Esempio evoluto lett. riga 2/2

```
void leggi_riga(istream &is, char *riga)
{
 // consuma eventuali sequenze di newline,
 // per evitare errori nella lettura di
 // una riga che si sta per effettuare
  while (is.peek() == '\n')
          is.get() ;

  is.get(riga, MAXLUN) ; // legge una riga
  is.get() ; // consuma il newline che
          // segue la riga appena letta
}
```

Lettura in un buffer 2

`read(char *buffer, int n)`

- Legge **n** byte e li copia nell'array **`buffer`**
- **Non è previsto alcun delimitatore, né aggiunto alcun terminatore**
  - Funzione di più basso livello rispetto all'ultimo uso visto della **`get`**
  -

`gcount()`

- Ritorna il numero di caratteri letti nell'ultima operazione di lettura

Scrittura non formattata

`put(char c)`

- Trasferisce un byte (il contenuto di **`c`**) sullo *stream* di uscita

`write(const char *buffer, int n)`

- Trasferisce i primi **`n`** byte dell'array **`buffer`** sullo *stream* di uscita
- Non è aggiunto alcun terminatore

File di testo e file binari 1/2

- Se si effettuano solo letture e scritture formattate su uno stream, si dice che lo si sta usando in **modo testo**
- In maniera simile, come già sappiamo, un file i cui byte sono da interpretarsi come codici di caratteri si definisce un **file di testo**
- Altrimenti si usa tipicamente la denominazione **file binario**

File di testo e file binari 2/2

- Per lavorare con file binari sono estremamente comode le letture/scritture non formattate, perché permettono appunto di ragionare in termini di pure sequenze di byte
- Leggere le slide relative alla memorizzazione delle informazioni della esercitazione 10
  - Fino all'esercizio *scrivi_leggi_array.cc* **escluso**

Riassumendo

- In qualsiasi modo sia stato scritto, un file rimane comunque solo una sequenza di byte
- Il fatto che sia un file di testo o un file binario è solo una questione di come si vuole o si deve interpretare tale sequenza di byte

Puntatori ed array

- Estendiamo le nostre conoscenze
- Come sappiamo il tipo puntatore
- **`<tipo> *`**
- memorizza indirizzi
- Come mai possiamo passare un array come argomento attuale nella posizione corrispondente ad un parametro formale di tipo **`<tipo> *`** ?
- Perché, come sappiamo, passare il nome di un array è equivalente a passare l'indirizzo del primo elemento dell'array

Esempio con tipo *char*

```
 void fun(const char *a)
{
 ofstream f("nome") ;
 // trasferiamo due elementi,
 // ossia due byte dell'array a
 f.write(a, 2) ;
}

main()
{
 char b[3] = {14, 31, 66} ;
 fun(b) ; // passo l'indirizzo di b
}
```

Oggetti e puntatori

- Il tipo **`char *`** memorizza indirizzi
  - Possiamo scriverci dentro l'indirizzo di qualsiasi oggetto, dinamico o non dinamico, non solo quindi di un array dinamico
  - In particolare, possiamo anche scriverci dentro l'indirizzo di oggetti di **tipo diverso** da array di caratteri

Trasferimento oggetti generici

- Le funzioni di ingresso/uscita non formattate si aspettano però solo array di caratteri
- Per usare tali funzioni dobbiamo perciò convertire il tipo dell'indirizzo di un oggetto diverso da un array di caratteri mediante:

`reinterpret_cast<char *>(<indirizzo>)`

  - Se appropriato si può anche aggiungere il *const*
  - Proviamo a vedere il significato logico di tale conversione

Oggetti in memoria 1/3

- Consideriamo ad esempio un array di interi di 3 elementi:

- Supponendo che ogni elemento occupi 4 byte, l'array in memoria sarà fatto così:

- Si tratta di una tipica sequenza di byte

Oggetti in memoria 2/3

- Tale sequenza di byte ha qualche differenza intrinseca rispetto ad una qualsiasi altra sequenza di byte di pari lunghezza?

Oggetti in memoria 3/3

- No
- Prendendo in prestito dal C/C++ il termine array, possiamo dire che una sequenza di byte non è altro che un **array di byte**
- Ma sappiamo che un byte può essere rappresentato esattamente su di *char*
- Quindi, qualsiasi sequenza di byte può essere rappresentata con array di *char*

Significato conversione

- Pertanto, mediante l'espressione

`reinterpret_cast<char *>(<indirizzo_oggetto>)`

  - diciamo: “Reinterpreta l'indirizzo dell'oggetto come l'indirizzo a cui inizia una una sequenza di byte”
  - Da un punto di vista logico vogliamo reinterpretare come una pura sequenza di byte il contenuto della memoria a partire dall'indirizzo dell'oggetto
- Rimane il problema di sapere la lunghezza di tale sequenza di byte
  - Non solo, in generale potremmo voler trasferire una sequenza di byte, ossia un array di caratteri, relativa solo ad una porzione dell'intero oggetto

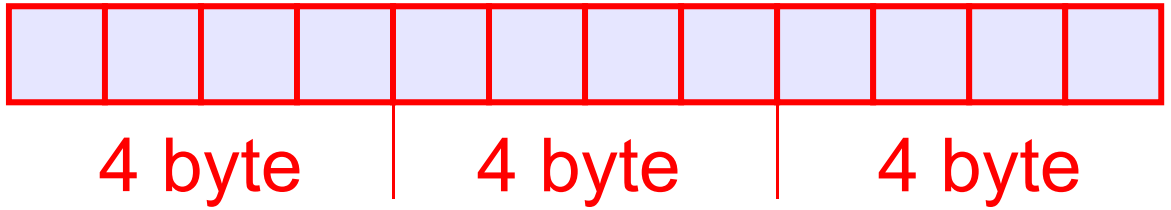
Dimensioni in byte 1/2

- Dato un generico array di elementi di qualche tipo, possiamo calcolare la lunghezza della sequenza di byte occupati da tutti o da una parte degli elementi dell'array nel seguente modo
- Utilizziamo l'operatore **`sizeof`** per conoscere le dimensioni di ogni elemento, e moltiplichiamo per il numero di elementi

Dimensioni in byte 2/2

- Consideriamo di nuovo un array di interi di 3 elementi e supponiamo che'operatore sizeof ci dica che ogni elemento occupa 4 byte:

- L'array in memoria occupa 4x3=12 byte



Scrittura intero array

```
 void scrivi_array_su_file(const int *a)
{
  ofstream f("file_destinazione") ;
  f.write(
     reinterpret_cast<const char *>(a),
  sizeof(int) * 3
  ) ;
}

main()
{
  int b[3] = {1, 2, 7} ;
  scrivi_array_su_file(b) ;
}
```

Scrittura su file binari

- Cosa abbiamo fatto?
- Abbiamo scritto nel file di nome *file_destinazione* una sequenza di byte uguale alla rappresentazione in memoria, byte per byte, dell'array di interi **b**
- *file_destinazione* sarà certamente un file binario

Scrittura di parte dell'array

```
 // scrive solo i primi due elementi
void scrivi_array_su_file(const int *a)
{
  ofstream f("file_destinazione") ;
  f.write(
     reinterpret_cast<const char *>(a),
  sizeof(int) * 2
  ) ;
}

main()
{
  int b[3] = {1, 2, 7} ;
  scrivi_array_su_file(b) ;
}
```

Esercizi

- Dalla decima esercitazione: *file/scrivi_leggi_array.cc*
- Leggere poi le successive slide fino all'esercizio *file_binario.cc* escluso

Domanda

- In base a quanto appreso finora
- Le funzioni di lettura e scrittura non formattate si possono utilizzare solo per leggere e scrivere file binari?
  - Ossia file in cui byte non sono da interpretare come codici di caratteri?

Risposta & domanda

- Decisamente no
- Come si può scrivere un file di testo mediante scritture non formattate?

Risposta

- Scrivendo i byte che rappresentano i codici dei caratteri desiderati
  - Un byte alla volta
    - Torneremo in generale sulla scrittura un byte alla volta fra qualche slide
  - O più byte alla volta, effettuando, ad esempio, delle *write* di interi *array* di caratteri
    - Come nell'esempio di tipo *char* che abbiamo già visto

Conclusione

- Mediante letture e scritture non formattate si possono leggere e scrivere anche file di testo
- Per leggere e scrivere file di test si utilizzano tipicamente letture e scritture formattate semplicemente perché queste ultime sono estremamente più comode
  - Progettate esattamente per leggere e scrivere agevolmente file di testo

Domanda

- Si possono scrivere gli elementi uno alla volta in un file mediante un ciclo di scritture non formattate?

Bufferizzazione 1/2

- Sì, ma è molto inefficiente
- Scriviamo invece tipicamente gli elementi uno alla volta nel caso di scritture formattate
- Non si perde efficienza in questo caso?

Bufferizzazione 2/2

- No, proprio perché, come già sappiamo, l'operatore di uscita bufferizza le operazioni

Passaggio indirizzo

- Come abbiamo visto, il nome di un *array* in una qualsiasi espressione denota l'indirizzo di un *array*
- Pertanto passare un *array* come parametro attuale equivale a passare l'indirizzo dell'*array*
- E se volessimo passare ad una delle funzioni di ingresso/uscita l'indirizzo di un oggetto diverso da un *array*?
  - Ad esempio un singolo intero, o un singolo oggetto di tipo struttura?

Operatore indirizzo

- In questo caso dovremmo utilizzare l'operatore indirizzo &
- Si tratta di un operatore unario prefisso
- Sintassi
- **`& <nome_oggetto>`**
- Semantica: ritorna l'indirizzo dell'oggetto passato per argomento
- Per capire come procedere dobbiamo considerare come è rappresentato l'oggetto in memoria

Generico oggetto in memoria

- Consideriamo un generico oggetto, per esempio di tipo strutturato

- Se l'oggetto occupa ad esempio 8 byte, allora in memoria si avrà la seguente sequenza di byte a partire dall'indirizzo dell'oggetto:

- Come già sappiamo, è una sequenza di byte come tutte le altre, rappresentabile mediante un array di caratteri

Esempio con struct 1/2

```
main()
{
     struct part {char nome[10]; int tempo}
            mario ;

  strcpy(mario.nome, "Mario") ;
  mario.tempo = 30 ;
  char * const p =
  reinterpret_cast<char *>(& mario) ;
}
```

Uso dell'operatore & per ritornare l'indirizzo dell'oggetto

Esempio con struct 2/2

- In **`p`** è finito l'indirizzo in memoria dell'oggetto struttura **`mario`**
- La conversione si è resa necessaria perché **`p`** punta ad oggetti di tipo diverso da **`part`**
- Se vogliamo scrivere l'oggetto su di un file
  - Come facciamo a copiare sul file solo l'effettivo numero di byte occupati da **`mario`**?

Risposta

- Utilizziamo l'operatore **`sizeof`**

Scrittura su file binari 1/2

```
main()
{
     struct part {char nome[10]; int tempo}
            mario ;

  strcpy(mario.nome, "Mario") ;
  mario.tempo = 30 ;
  char * const p =
  reinterpret_cast<char *>(& mario) ;
  ofstream f("dati.dat") ;
  f.write(p, sizeof(mario)) ;
}
```

Scrittura su file binari 2/2

- Cosa abbiamo fatto?
- Abbiamo scritto nel file *dati.dat* una sequenza di byte uguale alla rappresentazione in memoria, byte per byte, dell'oggetto **`mario`**
- *dati.dat* è certamente un file binario

Lettura da file binari

- Come facciamo a rimettere in memoria le informazioni salvate nel file?
  - Finire la decima esercitazione
- Prima di andare avanti è opportuno osservare che quanto fatto con oggetti di tipo **`struct`** è solo un altro esempio di lettura/scrittura da/su file binario
- Si potevano fare esempi con matrici o array di oggetti struttura, e così via ...

Accesso sequenziale e casuale

- Uno *stream* è **acceduto sequenzialmente** se ogni operazione interessa caselle dello *stream* consecutive a quelle dell'operazione precedente
- Si ha invece un **accesso casuale** ad uno *stream* se per una operazione si sceglie arbitrariamente la posizione della prima casella coinvolta
- Per **`cin`**, **`cout`** e **`cerr`** è possibile solo l'accesso sequenziale

Accesso casuale ai file

- La *casella* (ossia il byte) a partire dalla quale avverrà la prossima operazione è data da un *contatore* che parte da 0 (prima casella)
- Il suo contenuto può essere modificato con le funzioni membro

  `seekg(nuovo_valore)` per file in ingresso

  (la g sta per get)

  `seekp(nuovo_valore)` per file in uscita

  (la p sta per put)

Accesso casuale ai file

- Le due funzioni possono anche essere invocate con due argomenti
  `seekg(offset, origine)`
  `seekp(offset, origine)`
- L'origine può essere:
  `ios::beg` offset indica il numero di posizioni a partire dalla casella 0 (equivalente a non passare un secondo argomento)
  `ios::end` offset indica il numero di posizioni a partire dall'ultima casella (muovendosi all'indietro)

Lettura della posizione

- Per gli *ifstream* è definita la funzione
  `tellg()`
  Ritorna il valore corrente del contatore
- Per gli *ofstream* è definita la funzione
  `tellp()`
  Ritorna il valore corrente del contatore

Esercizio per casa

- Dato un file binario in cui sono memorizzati oggetti di tipo
-

```
struct  persona {
        char   codice[7];
            char   Nome[20];
            char   Cognome[20];
            int    Reddito;
            int    Aliquota;
    }  ;
```

ed assumendo che ogni persona abbia un codice univoco ...

Esercizio

- Scrivere una funzione che prenda in ingresso un oggetto *P* di tipo persona per riferimento, ed un *istream* che contiene la rappresentazione binaria di una sequenza di oggetti di tipo persona.
- La funzione cerca nello *istream* un oggetto con lo stesso valore del campo *codice* dell'oggetto *P* e, se trovato, riempie i restanti campi dell'oggetto *P* con i valori dei corrispondenti campi dell'oggetto trovato nel file.
- La funzione ritorna *true* in caso di successo, ossia se l'oggetto è stato trovato, *false* altrimenti

Soluzione

```
bool ricerca_in_file(persona &P, ifstream &f)
{
   bool trovato=false;
   while (true) {
      persona buf[1] ;
      f.read(reinterpret_cast<char *>(buf),
             sizeof(persona)) ;
      if (f.gcount() <= 0) // EOF o errore
            break ;
      if (strcmp(buf[0].codice, P.codice) == 0){
            P = buf[0] ;
            trovato = true ;
            break ;
      }
  }
  return trovato;
}
```

Osservazioni finali 1/2

- I file sono una delle strutture dati fondamentali per la soluzione di problemi reali
- Infatti nella maggior parte delle applicazioni reali i dati non si leggono (solamente) da input e non si stampano (solamente) su terminale, ma si leggono da file e si salvano su file

Osservazioni finali 2/2

- Spesso si rende necessario gestire in modo efficiente grandi quantità di dati su supporti di memoria di massa
  - Vedrete come negli insegnamenti di
  - “BASI DI DATI”
- Ricordare infine che la gestione dei file sarà sempre parte della prova di programmazione