Lezione 14

Stream, fstream e file

Input streams 1/2

- *stream*: sequenza di caratteri
- *istream*: meccanismo per convertire sequenze di caratteri in valori di diverso tipo
- *standard istream*: ***cin***
  - tipicamente associato al terminale da cui è fatto partire il programma
  - appartiene al *namespace std*
    - E' questo uno dei motivi per cui abbiamo aggiunto nei nostri programmi la direttiva

```
using namespace std ;
```

Input streams 2/2

- Come si leggono valori?
- Operatore >> (*leggi*, *estrai*)
- Input formattato ...

Input formattato

- Le cifre sono convertite in numeri se si leggono interi o reali
- I caratteri speciali sono saltati se si leggono singoli caratteri
  - Ad esempio è saltato il newline `'\n'`
- Gli spazi bianchi (spazio, tab, newline, form-feed, ...) sono saltati se si leggono stringhe

Stream state

- Ciascun (*i/o*)*stream* ha un proprio *stato*
  - Insieme di *flag* (valori booleani)
- Errori e condizioni non standard sono gestiti assegnando o controllando in modo appropriato lo stato

End Of File (*EOF*)

- Classica condizione che impedisce di effettuare ulteriori operazioni: leggere la *marca EOF* da uno stream di input
  - Nel caso in cui si stia effettivamente leggendo un file attraverso l'*istream*, vuol dire che si è raggiunta la fine del file
  - Nel caso di input da un terminale UNIX, l'*EOF* è generato se si preme *Ctrl-D* (su riga vuota)

Espressioni con >>

- *cin* e *cin>>...*, oppure *!cin* e *!(cin>>...)* sono espressioni
  - Es.:, *cin>>dim* è una espressione che ha un suo valore di ritorno
  - Ovviamente come sappiamo la valutazione di tale espressione comporta la lettura da *cin* mediante l'operatore >>, e quindi l'assegnamento alla variabile *dim* di un opportuno valore in base al contenuto (e come vedremo allo stato) del *cin*

Controllo stato istream

- Le precedenti espressioni si possono utilizzare dove è atteso un valore booleano, ed in tal caso il significato del loro valore è il seguente
  - Vero: la prossima operazione può aver successo perché lo stream è in stato *buono*
  - Falso: la prossima operazione fallirà perché lo stream è in stato *non-buono*
    - Il motivo per lo stato *non-buono* è che l'ultima operazione è fallita: formato errato dell'input oppure incontrato *EOF*

Stato istream

- Una volta in stato *non-buono*, lo *stream* ci rimane finché i flag non sono esplicitamente resettati
- Semplice istruzione per resettare lo stato dello *stream*:

  cin.clear() ;
- Operazioni di input su *stream* in stato non-buono sono operazioni nulle
- Bisogna resettare *prima* di effettuare la prossima operazione di input

Esercizio

- Scrivere un programma che, dopo aver letto da *stdin* una sequenza di numeri interi, stampi la somma dei valori letti
- La lunghezza della sequenza **non è nota a priori**, **nè comunicata prima** di iniziare ad immettere i numeri
- Soluzione nella prossima slide

Esercizio

```
#include <iostream>

using namespace std ;

main()
{
      int i, somma = 0 ;
      while (cin>>i)
            somma += i ;
      cout<<"Somma: "<<somma<<endl ;
      // se volessi continuare ad usare
      // il cin, dovrei prima invocare
      // cin.clear()
}
```

Operazioni di input nulle

- Come già detto, quando una operazione di input fallisce, è una vera e propria operazione nulla:
  - **nessun carattere** è **rimosso** dallo stream di input
  - il valore della variabile di destinazione è **lasciato inalterato**
- Esempio:

```
int i = 3 ;
cin>>i ; // se il cin è in stato di errore,
         // in i rimane 3 indipendentemen-
         // te dal contenuto dello stdin
```

Output streams

- *ostream:* meccanismo per convertire valori di vario tipo in sequenze di caratteri
  - Output formattato
- *standard output ostream* e *standard error ostream:* ***cout*** e ***cerr***
  - *ostream* tipicamente collegati al terminale da cui è fatto partire il programma
  - appartengono al *namespace std*

Operazioni di uscita 1/2

- Supponiamo per un momento che all'interno dell'oggetto *cout* vi sia del codice che scriva immediatamente su *stdout* ogni singolo carattere ad esso passato mediante l'operatore <<
- Questo comporterebbe la lettura separata di ciascuno di tali caratteri da parte della shell da cui abbiamo lanciato il programma
- La shell a sua volta passerebbe uno ad uno i i caratteri letti terminale, che si occuperebbe a sua volta di comunicare col sistema operativo per farli apparire (di nuovo uno alla volta)

Operazioni di uscita 2/2

- Ciascuna delle precedenti operazioni su singolo carattere avrebbe più o meno lo stesso costo se effettuata su un intera stringa
- Allora perché non cercare di effettuarle per stringhe di più di un carattere anziché carattere per carattere?
- Una possibilità sarebbe ad esempio quella di mandare su *stdout* una riga alla volta, ossia una stringa che ha un *newline* come ultimo carattere

Buffer 1/2

- A questo scopo potremmo immaginare che l'oggetto *cout* memorizzi temporaneamente i caratteri che gli vengono passati in un proprio array di caratteri, e li scriva effettivamente sullo *stdout* solo quando quest'array di caratteri arriva a contenere una riga
- Tutte le precedenti operazioni non sarebbero più effettuate per ogni singolo carattere, ma una riga alla volta
- Molto più efficiente

Buffer 2/2

- Tale array di caratteri è un esempio di buffer
- Si indica col termine **buffer** (memoria tampone) un array di byte utilizzato nelle operazioni di I/O
  - Vi si memorizzano temporaneamente le informazioni prime di spostarle nella destinazione finale
  - Il motivo principale per l'uso di un buffer è l'efficienza

Uscita bufferizzata

- Le operazioni di uscita con gli stream sono effettivamente tipicamente *bufferizzate*
- Ad esempio il passaggio dei caratteri da stampare allo *stdout* non avviene carattere per carattere, bensì i caratteri vengono spediti tutti assieme proprio quando si inserisce il *newline*

Buffer e incoerenza dell'uscita

- Si possono però avere problemi di incoerenza delle informazioni in uscita
  - Ad esempio se un programma fallisce dopo una scrittura su *cout* in cui non si è inserito il *newline*, i corrispondenti caratteri potrebbero non essere mai passati allo *stdout*
- Vederemo a breve un problema simile con le scritture su file

Svuotamento del buffer

- Si può scatenare lo svuotamento del buffer anche senza l'invio del *newline*
  - *endl* inserisci un newline e svuota il buffer di uscita (la soluzione con il newline che conosciamo già)
  - *flush* svuota il buffer di uscita (senza aggiungere alcun carattere)

    Es.:

```
cout<<"Prova"<<flush ;
```

Due argomenti extra

- I prossimi due argomenti, ossia rimuovere i caratteri da *stdin* e formattare l'output, non saranno argomento d'esame

Rimuovere caratteri da *stdin*

- Si possono rimuovere incondizionatamente caratteri da un istream con la seguente funzione
  - *cin.ignore()* ignora, ossia rimuove, il prossimo carattere da *stdin*
- Vediamone l'uso con un esempio

Soluzione non sicura

```
int main()
{
   int *p ; // attenzione, per ora contiene un valore casuale !!!
   int dim = -1 ;

   do { // immissione dimensioni array
      cout<<"Dimensioni array? " ;
      cin>>dim ;
   } while (dim < 0) ;

   p = new int[dim] ; // allocazione memoria

   ...
}
```

Che succede se si immette un carattere anziché una cifra?

Una soluzione sicura

```
int main()
{
   int *p ; // attenzione, per ora contiene un valore casuale !!!
   int dim = -1 ;

   do { // immissione dimensioni array
      cout<<"Dimensioni array? " ;
      while(!(cin>>dim)) {
            cin.clear() ;
            cin.ignore();
            cout<<"Devi immettere un numero: " ;
      }
   } while (dim < 0) ;

   p = new int[dim] ; // allocazione memoria

   ...
}
```

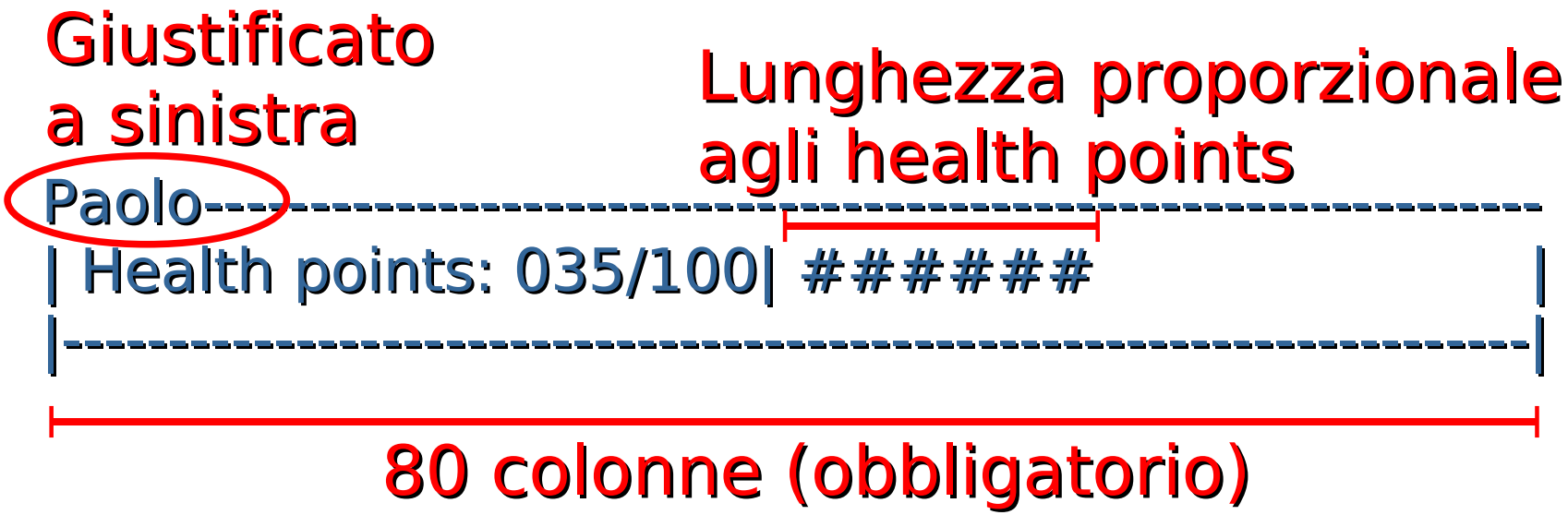
Però non esce in caso di EOF!
La soluzione è lasciata al lettore ...

Formattazione dell'output

- Come già detto i seguenti argomenti di formattazione dell'output non saranno argomento di esame

Esempio di output formattato

What's your name? Paolo
Health (in hundredths)? 35
Welcome to GOTA, Paolo. And good luck!



>

Formattazione dell'Output

- La formattazione è controllata da un insieme di flag e valori interi
- Semplice interfaccia per assegnare tali valori: *funzioni* dedicate e *manipolatori*

setprecision

- *cout.setprecision(int n)*

  Setta il massimo numero di cifre per un numero in virgola mobile

  - l'effettivo output dipende dal formato (generale, scientifico, fisso)

  - l'effetto è *persistente*: influenza tutte le prossime operazioni di uscita, fino alla prossima evetuale chiamata di *setprecision*

Manipolatore

- Operazione che modifica lo stato, da passare agli operatori di ingresso/uscita allo stesso modo degli oggetti che si da scrivere/leggere

- Esempi (già visti) di manipolatori che non prendono argomenti:
  - *flush* svuota il buffer di uscita
  - *endl* inserisci un newline e svuota il buffer di uscita

Manipolatori con argom. 1/2

- Spesso si vuole riempire con del testo predefinito un certo spazio su una linea

- *cout<<...<<setw(int n)<<...*

  Setta il minimo numero di caratteri per la prossima operazione di uscita

- *cout<<...<<setw(...)<<setfill(char c)<<...*

  Sceglie il carattere in *c* come carattere di riempimento

Manipolatori con argom. 2/2

- Per usare manipolatori che prendono argomenti bisogna includere:

  #include <iomanip>

Stampa dello stato del gioco

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

using namespace std ;

int main()
{
  int punti_salute = 35 ;

  cout<<left<<setw(80)<<setfill('-')<<"Paolo"<<endl ;
                              cout<<"|              Health              points:
    "<<right<<setw(3)<<setfill('0')<<punti_salute<<"/100 | " ;
  int num_asterischi = punti_salute*51/100 ;
  cout<<setw(num_asterischi)<<setfill('#')<<"" ;
  cout<<setfill(' ') ;
  cout<<setw(53 - num_asterischi)<<right<<" |"<<endl ;
  cout<<setw(80)<<setfill('-')<<""<<endl ;

  return 0 ;
}
```

- Torniamo agli argomenti che saranno oggetto d'esame ...

Definizione di nuovi *stream*

- *cout*, *cerr*, *cin* sono già pronti all'uso quando un programma parte
- Sono creati automaticamente ed associati allo *stdout*, *stdin* e *stderr* del programma
- Però possiamo anche creare i nostri *stream*
  - Alla creazione di uno *stream* dobbiamo specificare l'oggetto a cui è associato
  - Un tipico oggetto a cui associare uno *stream* è un *file*

fstream

- I seguenti tipi di *stream* sono da associare ai file, e sono supportati direttamente dalla libreria standard del C++ (non da quella del C)
- *ifstream*: file stream di ingresso (lettura)
- *ofstream*: file stream di uscita (scrittura)
- *fstream*: file stream di ingresso/uscita
- Presentati in <[*i* | *o*]*fstream*> o in <*fstream*> (tutti e tre)

Modello di file

- Un file è visto come una sequenza di caratteri (byte) che, come vedremo, potrà essere letta attraverso un *ifstream* (o uno *fstream* opportunamente inizializzato) o modificata attraverso un *ofstream* (o di nuovo uno *fstream* opportunamente inizializzato)

Associazione a file

- Un (*i*|*o*)*fstream* viene associato ad un file mediante un'operazione chiamata **apertura** del file
  - Da quel momento in poi tutte le operazioni di ingresso/uscita fatte sullo *stream* si tradurranno in identiche operazioni sul contenuto del file
  - E' il sistema operativo che si occuperà di tutti i dettagli (che variano da sistema a sistema) necessari per eseguire le operazioni sulla macchina reale

Associazione a file

- Come nome del file si può indicare tanto un percorso assoluto che un percorso relativo
- Esempio di percorso assoluto:
  `/home/paolo/dati.txt`
  File di nome *dati.txt* nella cartella */home/paolo*
- Esempi di percorsi relativi (il file è cercato nella cartella corrente):
  `paolo/dati.txt`
  File di nome *dati.txt* nella sottocartella *paolo* della cartella corrente

  `dati.txt`
  File di nome *dati.txt* nella cartella corrente

Associazione a file

- E' possibile aprire più di un file contemporaneamente
- L'apertura di un file può fallire per diversi motivi
  - Ad esempio se si tenta di aprire in lettura un file inesistente
- E' opportuno controllare sempre l'esito dell'operazione di apertura prima di utilizzare un (*i*|o)*fstream*

Apertura file 1/3

- Un file è aperto in input definendo un oggetto di tipo *ifstream* e passando il nome del file come argomento

```
ifstream f("nome_file") ;
if (!f) cerr<<"l'apertura è fallita\n" ;
```

- Un file è aperto in output definendo un oggetto di tipo *ofstream* e passando il nome del file come argomento

```
ofstream f("nome_file") ;
if (!f) cerr<<"l'apertura è fallita\n" ;
```

Apertura file 2/3

- Un file può essere aperto per l'ingresso e/o l'uscita definendo un oggetto di tipo *fstream* e passando il nome del file come argomento
- Deve essere fornito un secondo argomento *openmode*
  - *ios_base::in* oppure *ios_base::out*
- Se non esiste, un file aperto in scrittura viene creato, altrimenti viene troncato a lunghezza 0

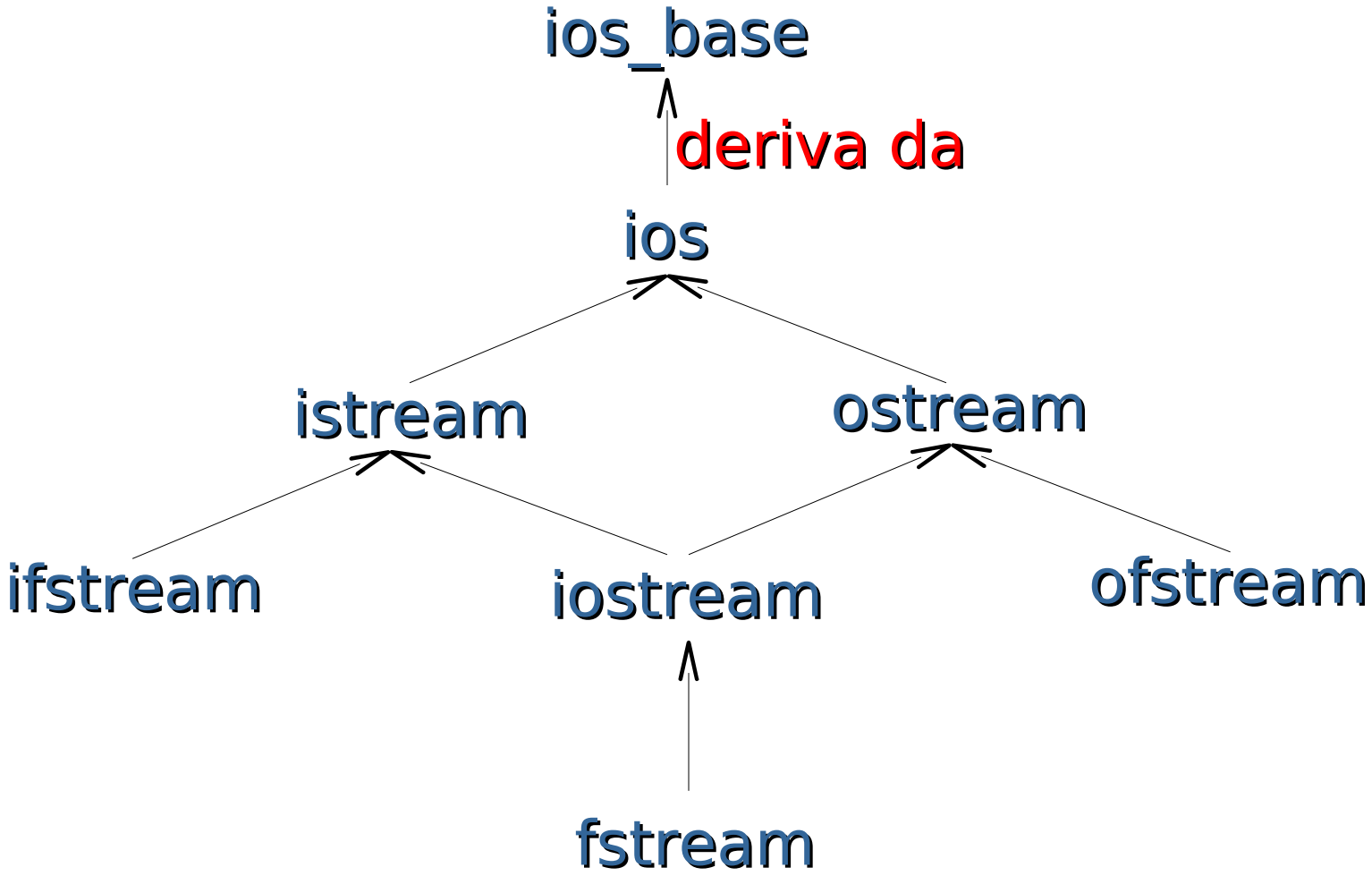
Apertura file 3/3

- Esempi:

```
// file aperto in ingresso
fstream f("nome_file", ios_base::in) ;
if (!f) cerr<<"apertura fallita\n" ;

// file aperto in uscita
fstream f2("nome_file", ios_base::out) ;
if (!f) cerr<<"apertura fallita\n" ;
```

Gerarchia degli stream



Conseguenza immediata

- Si possono usare tutti gli operatori, i flag di stato, e le funzioni di utilità per la formattazione viste per gli *stream* di ingresso/uscita standard

- Quindi si può quindi controllare lo stato di un oggetto (*i*|*o*)*fstream* `f` usando il suo identificatore in una espressione condizionale
  Esempio:

```
if (!f)
      cerr<<"La precedente operazione è"
          <<"fallita"<<endl ;
```

Informazioni aggiuntive

- Come visto negli esempi, con oggetti di tipo **`ifstream`**, **`ofstream`** o **`fstream`** si controlla il successo o meno dell'apertura controllando lo stato
- Più modi possono essere combinati mediante l'operatore |
- *in*: input, *out*: output, *app*: append (si continua a scrivere a partire dal fondo)

  ofstream f("nome_file", ios_base::app) ;

Bufferizzazione uscita

- Per gli stessi motivi di efficienza visti per gli *ostream* collegati allo *stdout*, anche le operazioni di uscita su *ofstream* (oppure *fstream* inizializzati in scrittura) sono tipicamente bufferizzate
- Quindi, a meno di passare ad esempio i manipolatori *endl* e/o *flush* non è garantito che una certa scrittura sia immediatamente effettuata sul file associato

Chiusura file

- Un file può essere chiuso invocando la funzione *close()* sullo *stream* ad esso associato
  Es.: `f.close() ;`

- Un file è comunque chiuso implicitamente alla distruzione dello *stream* associato

- La chiusura (esplicita o implicita) di un file è importante perché solo all'atto della chiusura ne è garantito l'effettivo aggiornamento (svuotamento dei buffer)

open

- Si può anche aprire un file invocando la funzione *open* su uno *stream* non ancora associato ad alcun file (non inizializzato o deassociato mediante *close*)
  - Per brevità non vedremo la *open* in queste lezioni

Esercizio

- *file/file.cc*

Domanda

- Siete riusciti a risolvere l'esercizio in modo completo?
- Probabilmente no, perché l'operatore >> ha saltato i caratteri `'\n'`
- Vedremo a breve la soluzione mediante I/O non formattato

Esercizio

- Leggere da un file di testo *dati.txt* una sequenza di numeri interi di al più 100 elementi, finché non si trova il primo elemento uguale a 0. Memorizzare tutti i numeri letti in un vettore.

Soluzione

```
main()
{
  int vett[100] ;
  ifstream f("dati.txt");
  if (!f)
        cerr<<"Errore di apertura file\n";
   else
        for (int i = 0 ; i < 100 && f>>vett[i]
                    && vett[i] != 0 ; i++)
                    ;
}
```

Esercizi (senza soluzione)

- ESERCIZIO 1: Leggere da un file di testo *dati.txt* una sequenza di numeri interi terminata da 0. Memorizzare in un vettore tutti i numeri negativi.

- ESERCIZIO 2: Leggere da un file di testo *dati.txt* una sequenza di numeri interi terminata da 0. Memorizzare in un vettore tutti i numeri compresi tra –30 e +30 escluso lo 0. Ordinare il vettore in modo crescente e stampare tutti i numeri positivi.

- ESERCIZIO 3: Leggere da un file di testo *dati.txt* una sequenza di caratteri terminata da *. Memorizzare in un vettore tutti i caratteri alfabetici.

Esercizio

- Scrivere in un file di testo *valori_pos.txt* tutti i numeri strettamente positivi di un vettore contenente N valori interi, con N definito a tempo di scrittura del programma.
  Alla fine, inserire il valore -1 come terminatore.

Soluzione

```
main()
{
  const int N = 10 ;
  int vett[N] ;
    ofstream f("dati.txt");
    if (!f)
        cerr<<"Errore di apertura file\n";
     else {
        for (int i=0; i<N; i++)
                    if (vett[i]>0)
                f<<vett[i];

                f<<-1 ;
           }
 }
```

Esercizi (senza soluzione)

- ESERCIZIO 1: Scrivere in un file di testo “carat.txt” tutti i caratteri di una stringa letta da input. Terminare la sequenza di caratteri del file con *.

- ESERCIZIO 2: Leggere da un file di testo “dati_inp.txt” una sequenza di numeri interi terminata da 0, e copiare in un vettore solo gli elementi positivi. Copiare tutti i valori del vettore compresi fra 10 e 100 in un file di testo “dati_out.txt”.

- ESERCIZIO 3: Come l’esercizio 4.c. In più, stampare su schermo il contenuto del file “dati_out.txt”.

I/O formattato

- Gli operatori di ingresso/uscita visti finora, ossia >> e <<, interpretano il contenuto di uno *stream* come una sequenza di caratteri
- Traducono quindi valori in 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 come una mera sequenza di byte e non effettuano alcuna trasformazione di alcun tipo
  - Trasferiscono semplicemente sequenze di byte da uno *stream* ad un array di byte (o un singolo byte), o da un array di byte (o un singolo byte) ad uno *stream*

Funzioni membro

- Le funzioni di lettura e scrittura non formattate che vedremo sono delle **funzioni membro**
- Per utilizzarle bisogna usare la notazione a punto
  - Se **f** è un *istream*, allora per usare ad esempio una funzione membro **`fun(char &)`**, bisogna scrivere **`f.fun(c) ;`**
    Es.:

```
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 le 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='\n')

read(char *buffer, int n)

gcount()
```

get()

- 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
- Esempio:

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

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:

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

get(char &c) 2/2

- Per semplicità, diciamo che 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:

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

Esercizio

- ***file/file_conta_linee.cc***

Lettura in un buffer

```
get(char *buffer, int n,
      char delimitatore='\n')
```

- Legge byte dallo stream di ingresso e li trasferisce in *buffer* aggiungendo 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'`

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, '-') ;
 ...
}
```

Lettura in un buffer 2

`read(char *buffer, int n)`

- Legge **`n`** byte e li memorizza nell'array **`buffer`**
- Non è previsto alcun delimitatore, né aggiunto alcun terminatore

`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 di **`buffer`** sullo *stream* di uscita
- Non è inserito 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
- Ricordare sempre però che un file rimane comunque solo una sequenza di byte, ed il fatto che sia un file di testo o un file binario è solo una questione di come è da 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 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>)
```

  - 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

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

  - Da un punto di vista logico vuol dire: “Interpreta come una 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

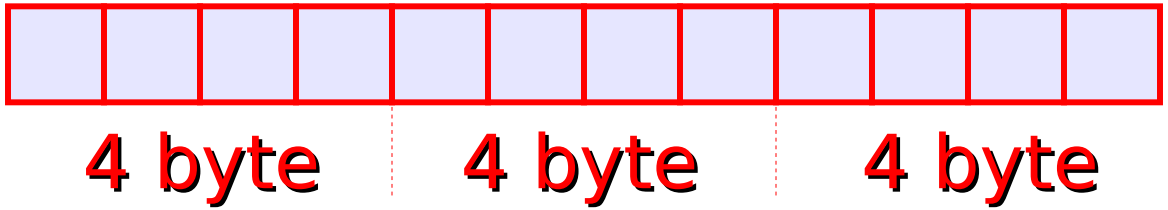
Dimensioni in byte 1/2

- Dato un generico array di elementi di qualche tipo, possiamo calcolare la lunghezza della sequenza di byte occupati da un certo numero di 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* la rappresentazione in memoria, byte per byte, dell'array di interi **b**
- *file_destinazione* sarà certamente un file binario
- Esercizio: *file/scrivi_leggi_array.cc*

Domanda

- Potevamo scrivere gli elementi uno alla volta nel file binario?

Bufferizzazione

- Sì, ma sarebbe stato molto inefficiente
- Abbiamo dovuto invece scrivere gli elementi uno alla volta nel caso del file di testo
  - Ma l'efficienza non si è persa, perché, come già detto, ci ha pensato l'operatore di uscita a bufferizzare le informazioni al posto nostro

Passaggio indirizzo

- Come abbiamo visto, il nome di un array in una espressione denota l'indirizzo di un array
- Pertanto passare un array come parametro formale 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
- Come sarà fatto 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 tipica sequenza di byte, 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) ;
}
```


Use 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
- Come facciamo ad accedere solo all'effettivo numero di byte occupati da **`mario`**?
- 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* la 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?
- *file_binario.cc*
- Prima di andare avanti è opportuno osservare che quanto fatto con un oggetto 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 definito ad **accesso sequenziale** se ogni operazione interessa caselle dello *stream* consecutive a quelle dell'operazione precedente
- Uno *stream* è definito ad **accesso casuale** se per una operazione può essere scelta arbitrariamente la posizione della prima casella coinvolta
- Per **`cin`**, **`cout`** e **`cerr`** è definito solo l'accesso sequenziale

Accesso casuale ai file

- La casella 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

Passaggio di *stream* 1/2

- Uno stream può essere passato per riferimento ad una funzione

  `cin` può essere passato come parametro attuale in corrispondenza di un parametro formale di tipo **`istream &`**

  `cout` può essere passato come parametro attuale in corrispondenza di un parametro formale di tipocome **`ostream &`**

Passaggio di *stream* 2/2

- Gli (*i*|*o*)*fstream* possono essere passati per riferimento dove sono attesi gli (*i*|*o*)*stream*

  un **`ifstream`** può essere passato come parametro attuale in corrispondenza di un parametro formale di tipo **`istream &`**

  un **`ofstream`** può essere passato come parametro attuale in corrispondenza di un parametro formale di tipo **`ostream &`**

Esempio

```
void stampa(ostream &o) {
 o<<"Stringa"<<endl ; }

void leggi(istream &i) {
 char s[100] ;
 i>>s ; cout<<s<<endl ; }

main()
{
 stampa(cout) ; // stampa su stdout
 ofstream f("nome_file.txt") ;
 stampa(f) ; // scrive nel file
 leggi(cin) ; // legge da stdin
 ifstream f2("nome_file.txt") ;
 leggi(f2) ;  // legge dal file
}
```

Esercizio

- 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_file(persona &P, istream &f)
{
  bool trovato=false;
    while (true) {
        persona buf[1] ;
        f.read(reinterpret_cast<char *>(buf),
               sizeof(persona)) ;
        if (f.gcount() <= 0)
              break ;
        if (strcmp(buf[0].Nome, P.Nome) == 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 grandi quantità di dati su supporti di memoria secondaria in modo molto efficiente
  - Gli informatici vedranno come negli insegnamenti di “BASI DI DATI”
- Ricordare infine che la gestione dei file sarà sempre parte della prova di programmazione