

# Lezione 11

---

Stream e file stream  
Input/output formattato

---

# Input stream

# Input stream

---

- *stream*: sequenza di caratteri
- *istream*: meccanismo per convertire sequenze di caratteri in valori di vari tipi
- *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 ;`**

# Scrittura su input stream

---

- L'input stream è riempito in qualche modo dal sistema in cui gira il programma
- Ad esempio, in caso di programma avviato da shell, dentro un terminale e senza redirectionamenti
  - Ciò che si scrive da tastiera finisce sull'input stream del programma
  - Ad esempio, se l'utente scrive la sequenza di caratteri `Ciao` e va a capo, sull'input stream del programma vi finisce:

'C'	'i'	'a'	'o'	'\n'
-----	-----	-----	-----	------

# Lettura da input stream

---

- Come si leggono valori?
- Operatore `>>`                      (*leggi, estrai*)
- Se la lettura ha successo, allora i caratteri letti per decidere il valore da immettere nella variabile sono eliminati dallo stream (ci torniamo sopra)
- Input formattato ...

# Input formattato

---

- Le cifre sono convertite in numeri se il tipo delle variabili in cui si scrive è intero o reale
- Gli spazi bianchi (spazio, *tab*, *newline*, *form-feed*, ...) sono tipicamente saltati
  - A meno di usare il manipolatore **`noskipws`**

# 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
- Una operazione che fallisce porta lo stream in stato di errore (stato **non buono**, come stiamo per vedere)

# End Of File (*EOF*)

---

- Classica condizione che causa il fallimento di una lettura e porta uno stream di input in stato di errore: leggere la *marca EOF*
- Nel caso in cui si stia effettivamente leggendo un file attraverso l'*istream*, si incontra tale marca solo se si è raggiunta la fine del file
- Nel caso di input da un terminale UNIX, si incontra l'*EOF* se l'utente preme *Ctrl-D* su una riga vuota



# Operazioni di input nulle

---

- Una operazione di input che 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
```

# 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 1/2

---

- 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: se la prossima operazione può aver successo perché lo stream è in stato *buono*
  - Falso: se la prossima operazione fallirà perché lo stream è in stato *non-buono*
    - Il motivo per lo stato *non-buono* è che l'ultima operazione effettuata è fallita: formato errato dell'input oppure incontrato *EOF*

# Controllo stato istream 2/2

---

- Esempi:

```
if (cin)
    cout<<"cin in stato buono"<<endl ;
```

```
if (! cin)
    cout<<"cin in stato di errore"<<endl ;
```

```
int i ;
if (! (cin>>i))
    cout<<"errore in lettura"<<endl ;
```

```
int j;
while (cin>>j) // valore di j significativo
    ...
```

# Stato istream

---

- Una volta in stato *non-buono*, lo *stream* ci rimane finché i flag non sono esplicitamente resettati
- Operazioni di input su *stream* in stato non-buono sono operazioni nulle
- Semplice istruzione per resettare lo stato dello *stream*:  
*cin.clear()* ;
- Bisogna resettare *prima* di effettuare la prossima operazione di input

# Domanda

---

- Cosa stampa il seguente frammento di codice se, quando lo si esegue, l'utente immette 5 da *stdin* ma l'oggetto *cin* è in stato di errore?

```
int i = 10 ;  
cin>>i ;  
cout<<i<<endl ;
```

- Stampa 10
- La lettura da *stdin* non viene effettuata: l'operazione è infatti nulla perché l'oggetto *cin* è in stato di errore

# 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()
    ...
}
```

# Domanda

---

```
#include <iostream>

using namespace std ;

main()
{
    int i, somma = 0 ;
    while (cin) {
        cin>>i ;
        somma += i ;
    }
    cout<<"Somma: " << somma << endl ;
    ...
}
```

Stampa correttamente la somma dei numeri inseriti fino a quando si preme Ctrl-D?

- No
- Quando si preme Ctrl-D la lettura fallisce
- Quindi nella variabile `i` rimane l'ultimo valore letto (o un valore casuale se non si è letto nulla in precedenza)
- Tale valore viene erroneamente sommato **prima** di controllare lo stato del `cin` per decidere se effettuare un'altra iterazione

---

# Output stream

# Output streams

---

- *ostream*: meccanismo per convertire valori di vario tipo in sequenze di caratteri
  - Output formattato: operatore <<
- *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*

---

# Compendio stream

# Compendio flussi di caratteri

---

- Ora possiamo approfondire e completare la nostra conoscenza dei flussi di caratteri e del comportamento degli operatori di ingresso/uscita formattato

# Uscita caratteri

---

- Cosa viene mandato sullo *stdout* dalla seguente istruzione?

```
cout<<'a' ;
```



# Risposta concettuale

---

- Concettualmente il carattere *a*
- Ma a più basso livello cosa viene mandato esattamente?

# Risposta di basso livello

---

- All'esecuzione dell'istruzione, sullo *stdout* viene inviato un byte contenente il codice del carattere *a*:

'a'

- Se si utilizza la codifica ASCII, il codice del carattere *a* è il numero 97, quindi sullo *stdout* viene inviato un byte contenente il numero 97:

97

- In effetti, a basso livello, lo *stdout* (come ogni flusso di caratteri) non è altro che una sequenza di byte, ciascuno contenente il codice di un carattere

# Domanda

---

- Alla luce di quanto detto finora, come mai
  - se il programma è invocato immettendone semplicemente il nome da riga di comando e premendo invio da una *shell*
  - allora quando viene eseguita l'istruzione

```
cout<<'a' ;
```

accade che appare il carattere *a* sul terminale?

# Risposta 1/2

---

- Perché la *shell*, prima di far partire il programma, aggancia lo *stdout* del programma ad un oggetto speciale del sistema operativo, oggetto tramite il quale il terminale legge i caratteri che deve far apparire
- In particolare il terminale è sempre in uno stato bloccato, in cui aspetta uno dei due seguenti eventi:
  - La segnalazione da parte di questo oggetto del fatto che è arrivato un nuovo carattere da far apparire sullo schermo 'virtuale' del terminale emulato
  - La segnalazione del fatto che è stato premuto un tasto sulla tastiera 'virtuale' del terminale emulato
- Quando accade uno dei due precedenti eventi, il terminale si sveglia, fa quello che deve fare e si blocca di nuovo in attesa del prossimo evento

# Risposta 2/2

---

- Quindi, ogni volta che si scrivono su tale oggetto speciale dei codici di caratteri, il terminale legge tali codici e fa apparire i corrispondenti caratteri
- Siccome la *shell* aggancia lo *stdout* del programma a tale oggetto prima di farlo partire, ogni istruzione del programma che scrive sullo *stdout*, scrive di fatto su tale oggetto

# Domanda

---

- Ma come mai, visto che il programma scrive semplicemente un numero sullo *stdout*, appare poi proprio il carattere *a*?

# Risposta

---

- Perché l'editor con cui si è scritto il programma, il compilatore con cui è compilato ed il terminale in cui è eseguito **utilizzano tutti la stessa codifica per il carattere**
  - Quando abbiamo scritto 'a', l'editor ha memorizzato nel testo del programma lo stesso codice che si aspetta il terminale per stampare il carattere *a*
  - Tipicamente entrambi usano la codifica ASCII
- Questo è sempre assicurato?
  - Purtroppo no

# Incongruenze codici

---

- Se si scrive un programma che stampa un carattere dal codice ASCII superiore a 127, il carattere che appare sul terminale può essere diverso da quello che appare nell'editor a parità di codice del carattere
  - Il terminale utilizza una tabella ASCII i cui codici al di sopra del 127 possono essere diversi da quelli utilizzati dall'editor
- Non ci interessiamo di questi problemi tecnici



# Ancora più a basso livello

---

- Abbiamo però detto che i byte sono solo sequenze di bit, tipicamente 8
- Allora anche un flusso di caratteri è una sequenza di sequenze di 8 bit ciascuna
- Quello che viene mandato sullo *stdout* dall'istruzione `cout<<'a' ;` è in effetti un byte contenente la sequenza di bit che corrisponde alla rappresentazione in base 2 del numero 97

01100001

# Domanda

---

- Cosa viene invece immesso sullo *stdout* dalla seguente istruzione?

```
cout<<'a'<<endl ;
```

# Risposta di basso livello

---

- Il codice del carattere *a* seguito dal codice del carattere speciale *newline*

'a'	'\n'
-----	------

- Assumendo che il *newline* sia rappresentato da codice 10 nella codifica ASCII, sullo *stdout* finisce

97	10
----	----

- Al più basso livello si tratta in effetti di due byte contenenti i seguenti bit:

01100001	00001010
----------	----------

# Stringhe e caratteri

---

- Passiamo ora alle costanti stringa
- Cosa viene immesso sullo *stdout* dalla seguente istruzione?

```
cout<<"Ciao"<<endl ;
```

# Risposta di basso livello

---

- La sequenza di codici dei caratteri che costituiscono la stringa, seguiti dal carattere speciale *newline*
- In particolare, all'esecuzione dell'istruzione, sullo *stdout* viene immesso:

'c'	'i'	'a'	'o'	'\n'
-----	-----	-----	-----	------

- Nel caso venga usata la tabella ASCII, numericamente si avrebbe:

67	105	97	111	10
----	-----	----	-----	----

- Per brevità in questa e nelle prossime slide non riportiamo più anche le sequenze di bit per ciascun byte, ma ci limitiamo alla notazione decimale

# Domanda

---

- Che differenza c'è tra l'effetto della seguente istruzione

```
cout<<"Ciao"<<endl ;
```

e quello della seguente istruzione?

```
cout<<'C'<<'i'<<'a'<<'o'<<endl ;
```

# Risposta

---

- Nessuna, entrambe mandano esattamente gli stessi caratteri sullo *stdout*

# Numeri e caratteri

---

- Supponendo che l'oggetto *cout* sia configurato per la stampa dei numeri in notazione decimale, cosa manda su *stdout* la seguente istruzione?

```
cout<<12 ;
```



# Risposta di alto livello

---

- Il numero 12

# Risposta di basso livello

---

- La sequenza di caratteri che rappresentano le cifre del numero 12 in base 10
- Ossia, in termini di sequenza di numeri su *stdout*:

'1'	'2'
-----	-----

- Che **non è uguale a**

1	2
---	---

- Ma, usando ad esempio la codifica ASCII, è uguale a:

49	50
----	----

# Domanda

---

- Che differenza c'è tra l'effetto della seguente istruzione  
`cout<<12<<endl ;`
- E quello della seguente istruzione?  
`cout<<'1 '<<'2 '<<endl ;`

# Risposta

---

- Nessuna, entrambe mandano esattamente gli stessi caratteri sullo *stdout*

# Rappresentazioni

---

- La sequenza di numeri (byte) mandata sullo *stdout* dall'istruzione `cout<<12 ;` è rappresentata in memoria da quali sequenze di bit?
  - Supponendo di utilizzare la codifica ASCII

# Domanda

---

- Come abbiamo visto, è rappresentata dalla sequenza:

00110001	00110010
----------	----------

- Tale sequenza di bit è uguale alla sequenza di bit utilizzata per rappresentare il numero 12 mediante un oggetto di tipo `int` in memoria?

# Risposta

---

- No
- In base 2, il numero 12 sarebbe 1100
- Ricordandoci che gli `int` occupano 4 byte sulle macchine attuali, uno dei modi in cui tale numero potrebbe essere rappresentato in memoria è

00000000	00000000	00000000	00001100
----------	----------	----------	----------

- Questo **NON è l'unico modo** in cui potrebbe essere rappresentato
  - In particolare l'ordine dei byte potrebbe essere diverso
  - Vedrete tutti i dettagli nell'insegnamento di Architetture dei calcolatori

# Riepilogando

- L'istruzione `cout<<12;` manda su *stdout* la sequenza di byte:

'1'	'2'
-----	-----

- Ossia, nel caso della codifica ASCII:

49	50
----	----

- Che, come sequenza di bit sarebbero:

00110001	00110010
----------	----------

- L'operatore di ha quindi convertito il numero 12 in una sequenza di byte/bit diversa da quella con cui il numero è rappresentato in memoria, ossia:

00000000	00000000	00000000	00001100
----------	----------	----------	----------



# Lettura caratteri

---

- Cosa accade invece quando si legge un carattere da *stdin* con le istruzioni  
`char a ; cin>>a ;`
- Vi sono due possibilità
  - Se sullo *stdin* sono già presenti dei caratteri, si consuma il primo della sequenza e si mette esattamente il suo valore all'interno della variabile **a**
  - Se sullo *stdin* non sono già presenti caratteri, il programma si blocca in attesa che finalmente vi arrivino
    - Non appena arrivano si fa la stessa cosa del caso precedente

# Terminale 1/2

---

- Come mai i caratteri immessi da terminale finiscono sullo *stdin* del programma?
- Perché la *shell*, prima di far partire il programma, aggancia lo *stdin* del programma ad un oggetto speciale del sistema operativo, sul quale il terminale spedisce i caratteri che vengono immessi da tastiera
  - Quindi, quando il programma legge un carattere da *stdin*, consuma il carattere in testa alla sequenza dei caratteri immessi su tale oggetto speciale dal terminale
    - Tale carattere **viene rimosso** dall'oggetto ed il prossimo carattere da leggere sarà quello che lo seguiva (se presente)

# Terminale 2/2

---

- Come mai i caratteri immessi dal terminale arrivano sullo *stdin* del programma solo quando si preme *invio*?
- Perché il terminale è tipicamente configurato per funzionare in una modalità, detta *canonica*, che prevede appunto la pressione del tasto *invio* per inviare i caratteri
  - Come abbiamo visto si può configurare però anche in altri modi

# Nota importante

---

- Quello che è importante capire è che il programma **si blocca solo se lo *stdin* è vuoto** quando viene eseguita l'istruzione di lettura di un carattere da *cin*
  - altrimenti legge il primo carattere disponibile senza bloccarsi

# Domanda

---

- Cosa ci assicura che, dato un carattere immesso dall'utente, il terminale inserirà sullo *stdin* proprio il codice corretto di quel carattere in base alla codifica usata nel nostro programma?

# Risposta

---

- Il fatto che il terminale, il compilatore usato per compilare il nostro programma e l'editor con cui abbiamo scritto il programma usino la stessa codifica
- Se le cose non stessero così sorgerebbero problemi
  - Tralasciamo di nuovo questi aspetti

# Lettura caratteri 1/2

---

- Tornando alla lettura di un carattere, se si eseguono le istruzioni

```
char a ; cin>>a ;
```

e sullo *stdin* vi sono i caratteri

'c'	'i'	'a'	'o'	'\n'
-----	-----	-----	-----	------

- Ossia, nel caso di codifica ASCII la sequenza di codici:

67	105	97	111	10
----	-----	----	-----	----

- Cosa finisce dentro la variabile *a* e cosa succede allo *stdin*?

# Lettura caratteri 2/2

---

- Nella variabile *a* finisce il codice del carattere *C*, ossia, se si usa la codifica ASCII, il numero 67
- Dallo *stdin* viene rimosso il primo byte, per cui vi rimane

'i'	'a'	'o'	'\n'
-----	-----	-----	------

- Una successiva lettura di un carattere leggerebbe il carattere *i* (senza che il programma si blocchi) e sullo *stdin* rimarrebbe

'a'	'o'	'\n'
-----	-----	------

- E così via



# Lettura numeri interi

- Consideriamo ora la lettura di un numero intero

```
int n ; cin>>n ;
```

e supponiamo che sullo *stdin* vengano immessi (o vi siano già) i caratteri

'1'	'2'	' '	'z'	'\n'
-----	-----	-----	-----	------

- Siccome il tipo della variabile `n` è `int`, l'operatore di ingresso consuma tutti i caratteri che trova sullo *stdin* finché li ritiene interpretabili come un numero
- In particolare, nel nostro esempio consuma i caratteri 1 e 2, e sullo *stdin* rimane

' '	'z'	'\n'
-----	-----	------

# Lettura numeri interi

---

- Dove finisce il numero letto?
  - Nella variabile `n`
- In che forma?
  - Dipende da come sono rappresentati i numeri di tipo `int` sulla macchina
  - Come si è visto una possibilità è

00000000	00000000	00000000	00001100
----------	----------	----------	----------

# Procedura

- Quindi, riepilogando, l'operatore di ingresso ha letto da *stdin* i byte

'1'	'2'
-----	-----

- Ossia, nel caso della codifica ASCII:

49	50
----	----

- Che, come sequenza di bit sarebbero:

00110001	00110010
----------	----------

- L'operatore di ingresso li ha quindi **interpretati** come il numero 12, e li ha memorizzati nella forma

00000000	00000000	00000000	00001100
----------	----------	----------	----------

# Caso di fallimento

- Consideriamo di nuovo la lettura di un numero intero  
`int n ; cin>>n ;`

e supponiamo che sullo *stdin* vengano immessi (o vi siano già) i caratteri

' '	'z'	'\n'
-----	-----	------

- Siccome il tipo della variabile `n` è `int`, l'operatore di ingresso salta (consumandolo) lo spazio sperando poi di trovare caratteri interpretabili come cifre di un numero intero
- Ma trova la `z` e l'interpretazione fallisce, per cui
  - la variabile `n` rimane inalterata e
  - sullo *stdin* rimane

'z'	'\n'
-----	------

# Operazioni di uscita 1/3

---

- 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 una operazione di scrittura per ciascuno di tali caratteri (sull'oggetto speciale del sistema operativo da cui il terminale legge cosa deve stampare)
- Sarebbero quindi sempre letti e stampati uno ad uno da parte del terminale

# Operazioni di uscita 2/3

---

- Se invece si scrivesse su tale oggetto un'intera stringa con una sola operazione
- tale operazione avrebbe più o meno lo stesso costo della scrittura di un singolo carattere (non vedremo i dettagli)
- anche la lettura e la successiva stampa sullo schermo da parte del terminale avrebbero più o meno lo stesso costo che avrebbero avuto se si fosse trattato di un singolo carattere

# Operazioni di uscita 3/3

---

- Allora perché non cercare di scrivere su *stdout* una stringa alla volta anziché un carattere alla volta?
- Una possibilità sarebbe ad esempio quella di mandare su *stdout* una riga alla volta
  - ossia una stringa che ha un *newline* come ultimo carattere

---

# Bufferizzazione uscita



- A questo scopo potremmo immaginare che l'oggetto *cout* memorizzi temporaneamente in un proprio array di caratteri nascosto i caratteri che gli vengono passati, e che 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

- Tale array di caratteri è un esempio di *buffer*
- Si indica col termine **buffer** (memoria tampone) un array temporaneo 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 è terminato forzatamente subito 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*

- *flush*           svuota il buffer di uscita (senza aggiungere alcun carattere)

Es.:

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

altrimenti c'è il manipolatore che già conosciamo e che aggiunge anche il *newline*:

- *endl*           inserirsi un *newline* e svuota il buffer di uscita

# 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?**



- Non si esce più dal ciclo, perché la lettura fallirà sempre

# 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() ; // da solo non basta!
            cin.ignore(); // ci vuole anche la ignore
            cout<<"Devi immettere un numero: " ;
        }
    } while (dim < 0) ;

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

    ...
}
```

Però non  
esce in  
caso di  
EOF!  
La soluzione  
completa  
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!

Giustificato  
a sinistra

Lunghezza proporzionale  
agli health points

Paolo

| Health points: 035/100| #####

80 colonne (obbligatorio)

>

# 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 eventuale 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 ...

---

# File stream

# 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***

- 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 `<fstream>` (tutti e tre assieme)

# 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
  - 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 quanto 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

---

- Un programma può aprire più di un file
- L'apertura di un file può fallire per diversi motivi
  - Ad esempio se si tenta di aprire in lettura un file inesistente
- A meno che si sappia quello che si fa, è opportuno **controllare sempre** l'esito dell'operazione di apertura prima di utilizzare un *(i|o)fstream*

# Apertura file 1/4

---

- 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 2/4

---

- Se non esiste, un file aperto in scrittura viene **creato**, altrimenti viene **troncato a lunghezza 0**
- Il contenuto precedente è perso

# Apertura file 3/4

---

- 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*

# Apertura file 4/4

---

- Esempi:

```
// file aperto in ingresso
```

```
fstream f("nome_file", ios_base::in) ;
```

```
if (!f) cerr<<"apertura fallita\n" ;
```

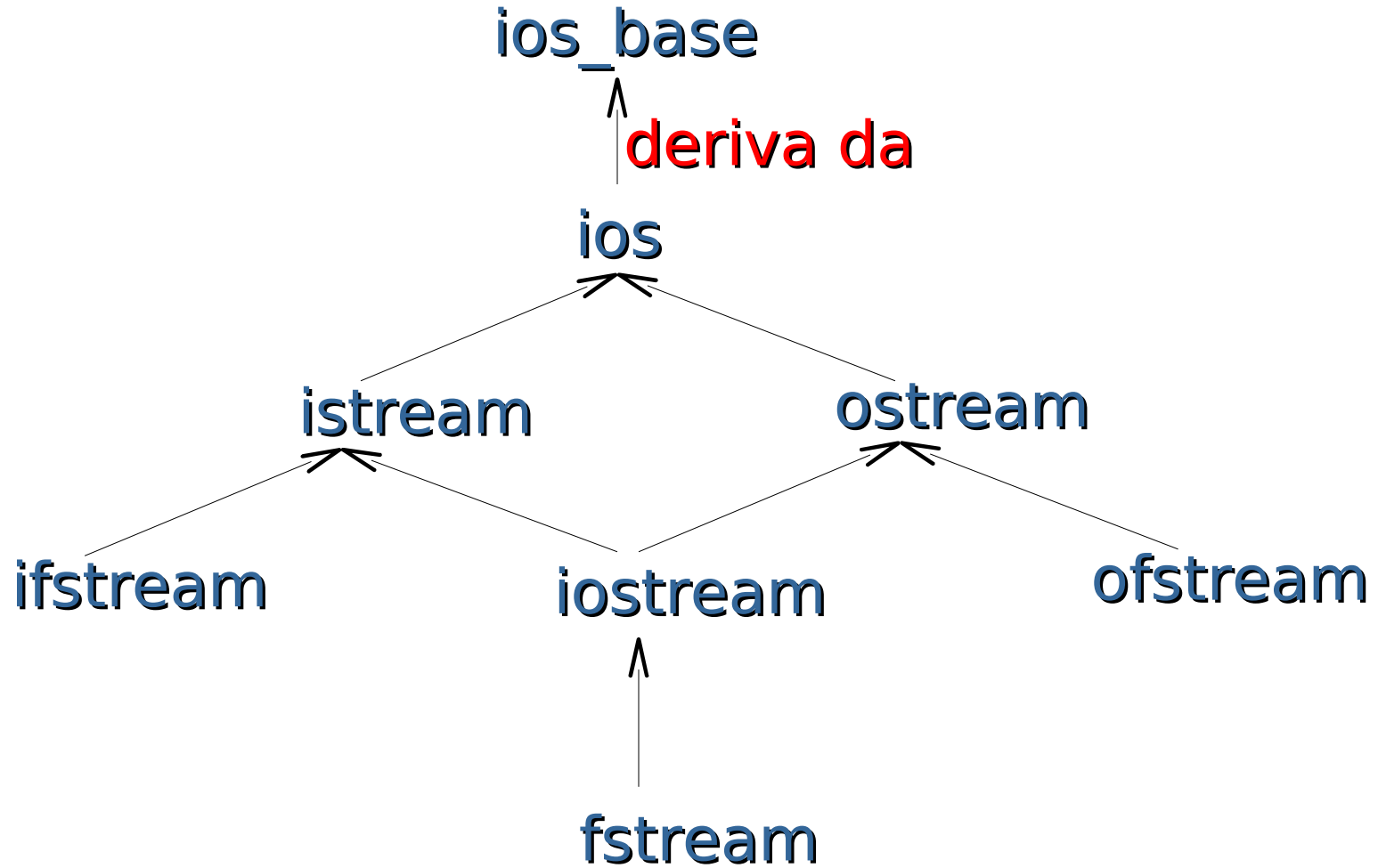
```
// file aperto in uscita
```

```
fstream f2("nome_file", ios_base::out) ;
```

```
if (!f) cerr<<"apertura fallita\n" ;
```

# Gerarchia degli stream

---



# Conseguenza immediata

---

- Per gli *fstream* 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 controllare lo stato di un oggetto *(i|o)fstream* £ usando il suo identificatore in una espressione condizionale

Esempio:

```
if (!f)
```

```
    cerr<<"La precedente operazione e'"  
    <<"fallita"<<endl ;
```



# 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)

- 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

---

- Scrivere un programma che:
  - 1) Crei un file di nome *Testo.txt*
  - 2) Chieda all'utente di inserire il contenuto del file, carattere per carattere mediante letture formattate (gestire opportunamente la comunicazione della fine dell'immissione dei caratteri da parte dell'utente)
  - 3) Chiuda il file
  - 4) Lo riapra in lettura
  - 5) Ne stampi il contenuto

# Soluzione

---

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

using namespace std ;

int main()
{
    //Creazione e apertura del file in scrittura
    ofstream f("Testo.txt");
    if (!f){
        cerr<<"Errore in creazione del file\n" ;
        return 1;
    }
    cout<<"Inserisci il contenuto del file "
        <<"(EOF per terminare l'input).\n";

    char c ;
    while(cin>>c) // lettura di un carattere da stdin
        f<<c ; // scrittura del carattere sul file
```

# Soluzione

---

```
// Chiusura file: garantisco l'avvenuta scrittura
f.close();

//Riapertura file in modalita' lettura
ifstream f2("Testo.txt") ;
if(!f2) {
    cerr<<"Errore in apertura file.\n" ;
    return 2;
}

cout<<"\nContenuto del file:\n" ;

while(f2>>c) // lettura di un caratree da file
    cout<<c ; // scrittura del carattere su stdout

return 0;
}
```

- Siete riusciti a risolvere l'esercizio in modo completo?
  - Probabilmente no, se non avete configurato l'oggetto *cin* per non saltare il carattere '`\n`'
- Vedremo in una prossima lezione anche la soluzione di questo esercizio mediante I/O non formattato

# Scrittura dal fondo

---

- Se si vuole aprire un file esistente in scrittura senza troncarlo a lunghezza zero, bisogna aprirlo nella cosiddetta modalità *append*
- Per farlo bisogna passare *ios\_base::app* come secondo parametro all'atto della definizione dell'*ofstream*
- Esempio:  

```
ofstream f("nome_file", ios_base::app) ;
```
- I byte che saranno inseriti sull'*ofstream* verranno **aggiunti a partire dal fondo del file**

# 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 operazione di scrittura sia immediatamente effettuata sul file associato



- 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 e stamparne il contenuto.

# 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 per casa

---

- 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”.

# Passaggio di *stream*

---

- Uno stream può essere passato per riferimento ad una funzione
- Il tipo di un parametro formale attraverso il quale passare un *istream* o un *ostream* per riferimento è ovviamente  
**istream &**  
oppure  
**ostream &**

# Passaggio *cin* e *cout*

---

- `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 tipo `ostream &`



# Passaggio di *(i|o)fstream*

---

- 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 scrivi(ostream &o) { // scrive sull'ostream o
    o<<"Stringa"<<endl ; }
```

```
void leggi(istream &i) { // legge dall'istream i
    char s ;
    i>>s ; cout<<s<<endl ; }
```

```
main()
{
    scrivi(cout) ; // stampa su stdout
    ofstream f("nome_file1.txt") ;
    scrivi(f) ; // scrive nel file
    leggi(cin) ; // legge da stdin
    ifstream f2("nome_file2.txt") ;
    leggi(f2) ; // legge dal file
}
```