# Input/output in C++: la libreria `iostream`
# Stringhe in C++: il tipo `std::string`

Pericle Perazzo

pericle.perazzo@iet.unipi.it

http://www.iet.unipi.it/p.perazzo/teaching/

http://lettieri.iet.unipi.it/mailman/listinfo/algoritmi_e_basi

8 marzo 2013


La libreria *STL* (*Standard Template Library*) è la libreria standard del C++, progettata dallo stesso inventore del C++: Bjarne Stroustrup. Include tutto ciò che riguarda l'output a video e l'input da tastiera, la gestione dei file, le strutture dati di base (liste, alberi, etc.) e gli algoritmi di base (binary search, quick sort, etc.). La parola *template* indica che fa uso estensivo del meccanismo dei template per creare strutture dati ed algoritmi generici.

**Output a video**
**Input da tastiera**

## Output a video – `std::cout`

```
1  #include <iostream>
2  int main()
3  {
4      int i = 120;
5
6      std::cout << "Hello, world!" << std::endl;
7      std::cout << "i = " << i << std::endl;
8  }
```

Per stampare a video, la STL usa l'oggetto `std::cout`. Questo codice mostra a video il messaggio "`Hello, world!`" (accapo) "`i = 120`" (accapo). L'operatore `<<` (put to) simboleggia la direzione dei dati che dalle variabili vanno al video.

Come si vede nell'esempio, le chiamate a `<<` possono essere concatenate su una sola riga di codice. Questo perché ogni chiamata restituisce il riferimento a `cout` per la chiamata successiva. `std::endl` sta per "end line" e viene usato per andare accapo.

Per usare `cout` bisogna include la libreria `<iostream>` (senza .h). Il prefisso `std::` è l'indicazione dello spazio di nomi (namespace) in cui si trovano tutte le strutture dati ed i metodi della STL. Si può omettere indicando "`using namespace std;`" fuori dal main.

## Input da tastiera – `std::cin`

```
1  #include <iostream>
2  int main()
3  {
4      int i;
5
6      std::cin >> i;
7      std::cout << "i = " << i << std::endl;
8  }
```

Analogamente, per l'input da tastiera si usa l'oggetto `std::cin`. Questo programma acquisisce un intero e poi lo stampa a video. Per `cin` si usa l'operatore duale al put to: il “get from” (>>). Anche questo simboleggia la direzione dei dati che dalla tastiera raggiungono le variabili.

## Vantaggi di `cout` e `cin`

- Sono *type-safe*:

Vecchio C

```c
#include <stdio.h>
int main()
{
	int a;
	char b[100];
	printf("%s", a);
}
```

```c
#include <iostream>
int main()
{
	int a;
	char b[100];
	std::cout << a;
}
```

- Sono *estensibili*:

```c
#include <iostream>
int main()
{
	int a;
	MyClass b;
	std::cout << a << b;
}
```

I vantaggi nell'usare gli oggetti `cin` e `cout` al posto delle funzioni standard del C `printf` e `scanf` sono molteplici.

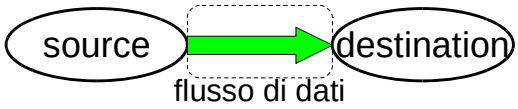
Prima di tutto, `cin` e `cout` sono type-safe. Con `printf` siamo obbligati a dichiarare il tipo della variabile da stampare mediante una speciale *wildcard* (`"%s"`). Cosa succede se il tipo della variabile non coincide con il tipo dichiarato con la wildcard? Il compilatore non dà errore ma il mio programma avrà un comportamento non prevedibile. Con `cout` invece il tipo della variabile viene dedotto automaticamente dalla variabile stessa. Una cosa analoga succede usando `cin` invece di `scanf`.

Un altro vantaggio è la possibilità di estendere gli operatori `<<` e `>>` ad accettare tipi definiti dall'utente. `printf` e `scanf` invece non possono essere estesi.

Un terzo vantaggio consiste nel fatto che `cin` e `cout` in realtà sono “stream”, e quindi possono essere trattati come generici flussi di dati.

## Il concetto di stream

- Gli *stream* (*flussi*) sono oggetti che astraggono una semplice comunicazione:
  - da una *sorgente* (produttore)
  - ad una *destinazione* (consumatore).




- Ad uno dei due estremi (sorgente o destinazione) c'è il programma stesso. All'altro estremo può esserci un dispositivo fisico, un file, una connessione di rete, un altro programma, etc. In `std::cin` la sorgente è la testiera, In `std::cout` la destinazione è il video.
- I dati che viaggiano su uno stream sono tutti dello stesso tipo. Noi tratteremo solo stream di *caratteri*.
- Uno stream può essere *chiuso* dalla sorgente. In questo caso viene inserito nel flusso di dati un particolare carattere detto *marca di fine stream*, o EOF (End of File).

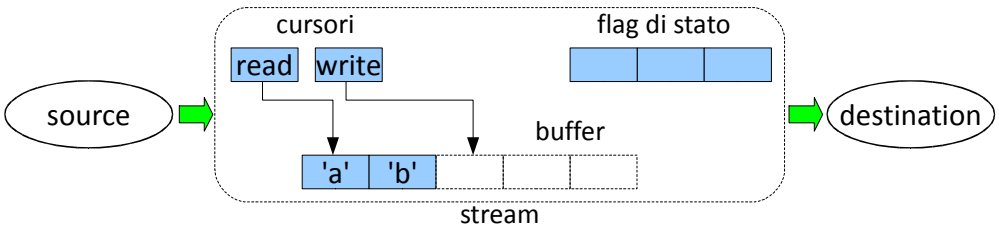
Uno *stream* (flusso) è un oggetto che astrae una semplice comunicazione da una sorgente ad una destinazione.Uno dei due estremi (sorgente o destinazione) è il programma che sto scrivendo. All'altro estremo può esserci un dispositivo fisico, un file, una connessione di rete, un altro programma, etc. Nel caso di `cin` la sorgente è la tastiera, nel caso di `cout` la destinazione è il video. Se il mio programma è la sorgente di dati, si parla di “output stream”. Se invece è la destinazione dei dati, si parla di “input stream”.

I dati che viaggiano su uno stream sono tutti dello stesso tipo. Noi tratteremo solo stream di *caratteri a 8 bit*.

Uno stream può essere *chiuso* dalla sorgente. In questo caso viene inserito un particolare carattere nello stream (diverso da tutti gli altri), chiamato “marca di fine stream” o EOF (End Of File). Tipicamente questo non viene fatto per gli stream `cin` e `cout`.

## Il concetto di stream

- Un flusso è implementato con un *buffer* e delle strutture dati di appoggio (contatori, indicatori di stato, etc.)

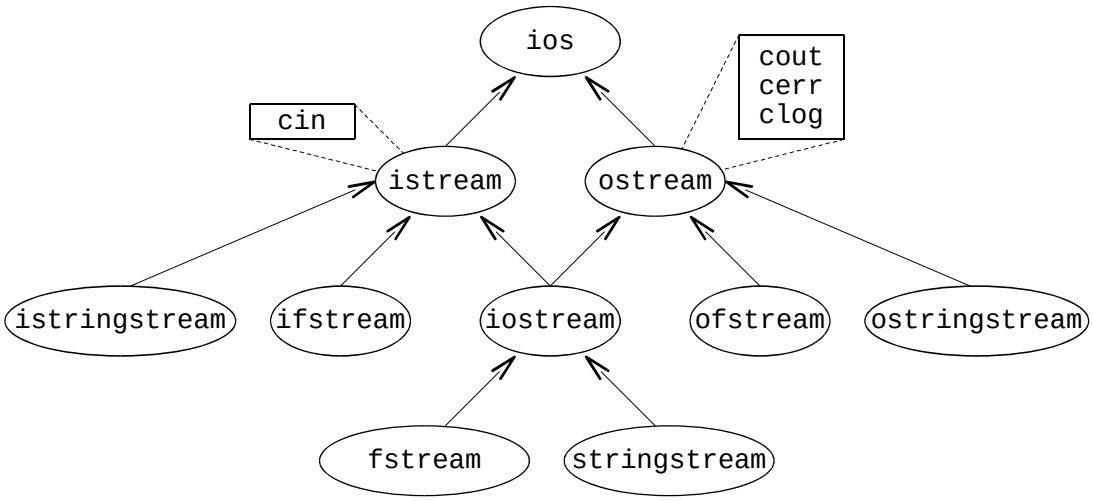



All'interno di uno stream c'è sempre un *buffer interno*. Quando la sorgente produce un dato, esso non raggiunge istantaneamente la destinazione ma viene memorizzato nel buffer. In un secondo momento, la destinazione lo estrae e viene cancellato dal buffer. Si dice che la comunicazione è *bufferizzata* (*buffered*).

Dire quindi che "`cout<<...` stampa su schermo" è inesatto. Bisognerebbe dire che "scrive sul buffer di uscita". L'effettiva stampa su schermo viene effettuata in un secondo momento.

All'interno dello stream ci sono anche due cursori di lettura e scrittura dal buffer e delle flag di stato che indicano l'esito (successo/errore) dell'ultima operazione.

## La gerarchia `std::ios`




Questa è la gerarchia (semplificata) della libreria iostream, che implementa gli stream in C++. La classe `ios` racchiude le funzionalità comuni. `istream` è un generico stream di input. `ostream` è un generico stream di output. l'oggetto `cin` è una speciale istanza di `istream`, mentre l'oggetto `cout` è una speciale istanza di `ostream`. Insieme a `cout` ci sono altri stream di uscita predefiniti: `cerr` per i messaggi di errore e `clog` per i messaggi di registro. Per default tutti e tre stampano a video ma con modalità diverse. In particolare, `cerr` è *unbuffered*, cioè non ha buffer interno. I dati quindi raggiungono istantaneamente il video.

`iostream` è un generico stream bidirezionale. Praticamente si comporta come un'unione di due stream: uno di input e uno di output.

`ifstream`, `ofstream` e `fstream` vengono usati per leggere/scrivere su file. `istringstream`, `ostringstream` e `stringstream` vengono usati per leggere/scrivere su una stringa in memoria.

## Vantaggi di `cout` e `cin`

- Possono essere trattati come stream *generici*:

```
1   using namespace std;
2   void myfun(ostream& out, int a, int b){
3         out << "a+2*b = " << a+2*b;
4   }
5   void main(){
6         // stampo su video:
7         myfun(cout, 1, 2);
8         // scrivo su file:
9         ofstream ofs(...);
10        myfun(ofs, 3, 4);
11        // scrivo su stringa:
12        ostringstream oss(...);
13        myfun(oss, 5, 6);
14  }
```

Come accennato sopra, il terzo vantaggio di `cout` e `cin` è che possono essere trattati come *stream generici*. In questo esempio, la funzione `myfun()` invia dei caratteri ad un generico stream di uscita e può essere usata sia per stampare a schermo, sia per scrivere su file, sia per scrivere su una stringa in memoria.

## La classe `std::ios`

- `std::ios` ingloba funzionalità comuni a tutti gli stream:
  - il buffer,
  - le flag per il controllo degli errori.
- I metodi `good()`, `fail()`, `bad()`, `eof()` permettono di accedere alle flag. Restituiscono rispettivamente:
  - `good()`: l'ultima operazione effettuata sullo stream è andata a buon fine,
  - `fail()`: ha riportato un errore,
  - `bad()`: ha riportato un errore irrecuperabile,
  - `eof()`: si è letta la marca di fine stream.
- In caso di errore lo stream si blocca.
- Il metodo `clear()` "resetta" le flag e sblocca lo stream.

`std::ios` ingloba funzionalità comuni a tutti gli stream, come la gestione del buffer, degli errori, etc.

Per fare controllo di errore sullo stream si usano i metodi `good()`, `fail()`, `bad()` ed `eof()`, che restituiscono dei valori booleani. `good()` dice se l'ultima operazione è andata a buon fine. `fail()`, al contrario, dice se l'ultima operazione ha incontrato un errore. `bad()` dice se questo errore è irrecuperabile Un errore recuperabile si ha, per esempio, quando il programma si aspetta da `cin` un numero e l'utente inserisce invece delle lettere. Un errore irrecuperabile si ha, per esempio, quando si verifica un guasto nello hardware o nel sistema operativo. `eof()` dice se l'errore è dovuto all'aver raggiunto la marca di fine stream.

Quando un'operazione causa un errore lo stream si blocca. Tutte le operazioni successive non avranno effetto e lo stream rimarrà nella condizione di errore fino a che non viene invocato il metodo `clear()`.

## Output – `std::ostream` e le sue derivate

- Contiene metodi per gli stream di *output*
- L'operatore << (*put to*) inserisce nel buffer dello stream
- È già definito per interi, double, char e stringhe C, etc:
  - `int i = 1; cout << i;` → interi
  - `double d = 3.14; cout << d;` → double
  - `char c = 'z'; cout << c;` → caratteri
  - `char s[] = "pippo"; cout << s;` → stringhe C
- Per i tipi utente occorre definire funzioni globali `friend`:

```
ostream& operator<<(ostream& out, const MyClass& m){...}
MyClass m; cout << m;
```

`std::ostream` è la classe base per tutti gli stream di output. L'operatore put to è definito per tutti i tipi base (`int`, `double`, `char`, `char[]`, etc.). Per i tipi utente occorre definire l'operatore come funzione globale, eventualmente `friend` della classe. Da `ostream` derivano `ofstream`, per scrivere su file, e `ostringstream`, per scrivere su una stringa memorizzata. Ogni operazione di scrittura ingrandisce opportunamente il file/la stringa. `ostringstream` è utile per preparare output formattato.

## Input – `std::istream` e le sue derivate

- Contiene metodi per gli stream di *input*
- L'operatore >> (*get from*) estrae dal buffer dello stream
- È già definito per interi, double, char e stringhe C, etc:
  - `int i; cin >> i;` → interi
  - `double d; cin >> d;` → double
  - `char c; cin >> c;` → caratteri
  - `char s[1000]; cin >> s;` → stringhe C
- Per i tipi utente occorre definire funzioni globali `friend`:

```
istream& operator>>(istream& in, MyClass& m);
MyClass m; std::cin >> m;
```

`std::istream` è la classe base per tutti gli stream di input. L'operatore `>>` è definito per tutti i tipi base (`int`, `double`, `char`, `char[]`, etc.). Per i tipi utente occorre definire l'operatore come funzione globale, eventualmente `friend` della classe. Da `istream` derivano `ifstream`, per leggere da file, e `istringstream`, per leggere da una stringa memorizzata.

## Funzionamento di >>

- Converte da codifica ASCII.
- Gli spazi iniziali vengono ignorati.

  Questi due input hanno lo stesso effetto:

```
10
```

```
    10
```

- Si possono inserire più campi sulla stessa riga.

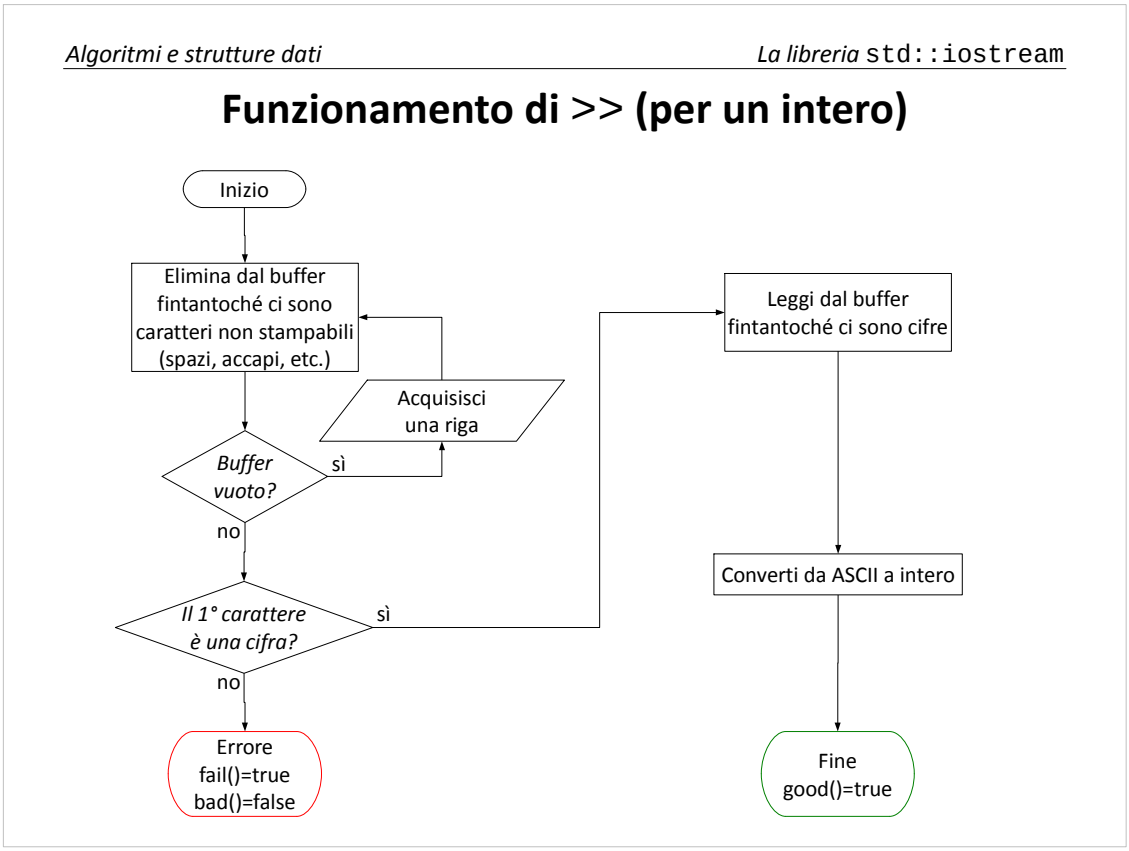
  Questi due input hanno lo stesso effetto:

```
10
100
1000
```

```
10 100 1000
```

- Se il buffer è vuoto, il >> si blocca fino a che non viene inserita una riga completa (fino all'accapo).

L'operatore get from (>>) è un po' complicato. È importante ricordarsi che: (1) gli spazi iniziali vengono sempre ignorati; (2) gli spazi delimitano campi diversi sulla stessa riga; (3) l'input viene acquisito una riga alla volta (fino alla pressione del tasto enter).




Questo diagramma di flusso descrive il funzionamento (semplificato) dell'operatore `>>` per un intero.

All'inizio si eliminano tutti gli spazi non stampabili dal buffer (per esempio gli spazi, gli accapi, i tab, etc.). Se il buffer è vuoto (com'è normalmente all'inizio del programma) si acquisisce una *riga* di caratteri. L'input viene acquisito e memorizzato nel buffer fino al carattere di terminazione riga (`'\n'`). Se il primo carattere stampabile non è una cifra (per esempio è una lettera) si produce un errore recuperabile (`fail()`=true). Altrimenti si leggono e si eliminano dal buffer tutte le cifre, finché non si incontra un carattere non-cifra (per esempio un accapo, uno spazio o una lettera). Dopodiché si convertono le cifre lette da ASCII a rappresentazione complemento a 2 e si termina l'algoritmo.

Si noti che l'algoritmo può terminare lasciando il buffer non vuoto. In questo caso i caratteri rimasti nel buffer verranno letti dalla prossima operazione di get from.

Algoritmi analoghi vengono eseguiti dalle get from degli altri tipi base: `char` `char[]`, `float`, `double`, etc. In particolare tutti scartano gli spazi iniziali e terminano la lettura del buffer al primo carattere *non interpretabile* (cioè uno spazio o una lettera nel caso di tipi numerici, etc.).

## Esempio – Funzionamento di >>

```
int i1;   int i2;
char s1[100];   int i3;
cin >> i1 >> i2 >> s1 >> i3;
```

Input da tastiera:

```
····10····100·····pippo
120pluto
```

Stato finale:

| Variabile: | Valore: |
|---|---|
| i1 | 10 |
| i2 | 100 |
| s1 | "pippo" |
| i3 | 120 |
| <buffer cin> | "pluto" |

Questo programma acquisisce due interi, una stringa e un terzo intero dalla tastiera. Se l'utente inserisce l'input in figura, le chiamate a get from si comporteranno come segue:

1.La prima chiamata a get from si blocca in attesa che l'utente inserisca la prima riga di input, fino all'accapo. Poi inizia a leggere il buffer. Elimina i 4 spazi iniziali, legge due cifre e mette il valore 10 in `i1`. Lascia nel buffer i restanti caratteri della riga.

2.La seconda chiamata non si blocca perché trova il buffer non vuoto. Elimina 4 spazi e legge 3 cifre. Mette il valore 100 in `i2`.

3.La terza chiamata elimina 6 spazi e legge una parola di 5 lettere, fino all'accapo. Mette il valore `"pippo"` in `s1`.

4.La quarta chiamata si blocca per ricevere una riga di input, poi legge dal buffer 3 cifre e mette il valore 120 in `i3`.

Alla fine dell'algoritmo, nel buffer di `cin` restano i caratteri “`pluto`”.

Si noti infine che la stringa è memorizzata su 100 caratteri. Questo può causare guasti di buffer overflow nel caso in cui l'utente inserisca più di 99 caratteri (più il terminatore di stringa che viene inserito automaticamente). Per evitare questo problema si può utilizzare il tipo `string` del C++, descritto in seguito.

## Lettura di un carattere stampabile / lettura di una carattere generico

- L'operatore >> applicato a un `char` legge un *carattere stampabile*.

```
char c1;
cin >> c1;
cout << c1;
```

```
····pluto
p
```

- Per leggere un carattere generico si usa `get()`.

```
char c1;
cin.get(c1);
cout << c1;
```

```
····pluto
·
```

L'operatore `>>` sui `char` acquisisce *caratteri stampabili*. Se invece si vuole acquisire caratteri generici (compresi spazi, accapi, etc.), bisogna usare la funzione `get()`.

## Lettura di una parola / lettura di una riga

- L'operatore >> applicato a una stringa legge una *parola* (di caratteri stampabili).

```
char s1[100];
cin >> s1;
cout << s1;
```

```
····pluto·····pippo
pluto
```

- Per leggere una *riga* (fino all'accapo) si usa `getline()`.

```
char s1[100];
cin.getline(s1, 100);
cout << s1;
```

```
····pluto·····pippo
····pluto·····pippo
```

L'operatore `>>` sulle stringhe acquisisce *parole*. Questo perché, come nel caso degli interi, vengono eliminati gli spazi iniziali e si legge il buffer finché ci sono caratteri diversi da spazio.

Il primo programma memorizza in `s1` solo la prima parola inserita, cioè `"pluto"`. Se si vuole acquisire delle intere stringhe fino all'accapo, senza eliminare gli spazi, bisogna usare il metodo `getline()`. `getline()` non ha problemi di overflow, in quanto prende in ingresso la dimensione della stringa. L'accapo finale viene eliminato dal buffer ma non viene copiato nella stringa `s1`.

## Esempio – Controllo di errore

```
1   #include <iostream>
2   using namespace std;
3   int main() {
4         int age;
5         cout << "How old are you?";
6         cin >> age;
7         if (cin.fail()) { // oppure: if(!cin)
8               cerr << "An error occurred" << endl;
9               exit(-1);
10        }
11        cout << "You are " << age << " years old" << endl;
12  }
```

Gestire gli stream di output è relativamente semplice. Questo perché abbiamo il controllo su ciò che scorre nello stream. Gestire gli stream di input invece è più problematico. Non possiamo prevedere a priori quanti e quali caratteri riceveremo. Quindi bisogna gestire vari casi di errore.

Questo programma di esempio prende da tastiera l'età dell'utente. Se qualcosa non va per il verso giusto (riga 7), per esempio l'utente inserisce delle lettere anziché delle cifre, viene stampato un errore a video.

Alcuni programmatori usano `if(!cin)` invece di `if(cin.fail())`. Il risultato è lo stesso.

## Esempio – Recupero da errore

```
1   int main(){
2       int age;
3       cout << "How old are you?";
4       cin >> age;
5       while(cin.fail()){
6           cout << "That's not a number!" << endl;
7           cin.clear();
8           cin.ignore(100, '\n');
9           cout << "How old are you? ";
10          cin >> age;
11      }
12      cout<<"You are "<<age<<" years old"<<endl;
13  }
```

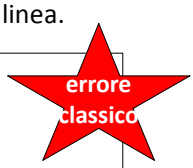
Questo programma è l'estensione di quello precedente, in cui si recupera l'errore logico di aver inserito lettere anziché cifre. In caso di errore stampiamo un avvertimento e ripetiamo la domanda.

Nel recupero da errore è importante ricordarsi di pulire le flag (`clear()`) e di pulire il buffer (`ignore()`). Se il buffer non viene pulito, i caratteri “spuri” che hanno fatto fallire l'operatore `>>` rimangono dentro il buffer e le prossime chiamate a `>>` falliranno nuovamente. Il metodo `cin.ignore(100,'\n')` elimina caratteri dal buffer fino al '\n' compreso o fino al 100esimo carattere.

## Bufferizzazione dell'output

- L'output è *bufferizzato*: ciò significa che il contenuto del buffer viene consegnato alla destinazione solamente quando ci viene scritto un carattere di fine linea.

```
1  long tm = time(NULL) + 5;
2  cout << "Wait 5 seconds... ";
3  while (time(NULL) < tm); // attendo 5 secondi...
4  cout << " ... done!" << endl;
```




- Lo scaricamento del buffer può essere imposto tramite il *manipolatore* `flush`.

```
1  long tm = time(NULL) + 5;
2  cout << "Wait 5 seconds... "<< flush;
3  while (time(NULL) < tm); // attendo 5 secondi...
4  cout << " ... done!" << endl;
```

Analogamente a quanto succede negli input stream, anche negli output stream il buffer viene gestito "per righe". Questo vuol dire che il buffer di `cout` viene effettivamente svuotato ed i caratteri compaiono a video solo quando si immette un carattere di fine linea (`'\n'` o `endl`). Facciamo un esempio.

Il metodo `time(NULL)` (libreria standard C) restituisce il tempo corrente del sistema in secondi. I cicli `while` in questi due frammenti di codice eseguono un'*attesa attiva*, cioè si bloccano per 5 secondi, testando continuamente la condizione del `while`.

Nel primo programma, il testo "`Wait 5 seconds...` " non appare quando ci si aspetterebbe, ma solo dopo l'attesa attiva, alla seconda chiamata a put to. La prima chiamata infatti, non spedendo caratteri di fine linea, e ha il solo effetto di scrivere sul buffer interno senza provocarne lo svuotamento.

Per imporre uno *svuotamento* del buffer si può usare il metodo `cout.flush()` o il *manipolatore* `flush`.

## Manipolatori

- Potrebbero richiedere di includere `<iomanip>`

| Manipolatore: | Significato: |
|---|---|
| `flush` | Scarica il buffer (solo output stream) |
| `endl` | Aggiunge un accapo e scarica il buffer (solo output stream) |
| `hex` | I successivi interi vengono formattati in base 16 |
| `oct` | I successivi interi vengono formattati in base 8 |
| `dec` | I successivi interi vengono formattati in base 10 |

```
cout << hex << 110 << dec << flush;
```

Output a video:

```
6e
```

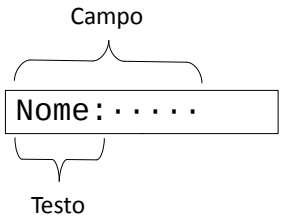
I *manipolatori* sono degli oggetti che possono essere usati come argomenti di `<<` e `>>` e che ne modificano il comportamento. Sopra ne vediamo alcuni esempi. `flush` svuota il buffer. `endl` aggiunge un carattere di fine linea e svuota il buffer. `hex` attiva la codifica esadecimale, `oct` quella ottale e `dec` ritorna alla codifica decimale.

## Output tabellato

- Come si fa ad allineare dei campi per fare un output in forma di tabella?

```
Nome:·····Cognome:···Matricola:·
Mario·····Rossi·····111111·····
Carlo·····Verdi·····222222·····
Francesco·Bianchi····333333·····
```

- Il *campo* ha lunghezza fissa. Il *testo* all'interno del campo ha lunghezza variabile (minore o uguale di quella del campo).




Per produrre un output tabellato si usa il concetto di *campo*. Il testo viene scritto all'interno del campo. Se la lunghezza del testo è minore di quella del campo vengono automaticamente aggiunti degli spazi per mantenere l'allineamento.

Se il testo è più lungo del campo, la tabella si disallinea. Quindi è importante controllare che la lunghezza del campo sia sempre maggiore o uguale a quella del testo.

## Output tabellato

- Manipolatori di campo (solo output stream)

| Manipolatore: | Significato: |
|---|---|
| `setw(w)` | Setta l'ampiezza del campo (default = 0) |
| `setfill(c)` | Setta il carattere di riempimento (default = *spazio*) |
| `left` | I successivi testi vengono allineati a sinistra nel campo |
| `right` | I successivi testi vengono allineati a destra nel campo (default) |

```
cout << left << setw(10) << setfill('*') << 300;
```

Output a video:

```
300*******
```

- Se la lunghezza del testo supera quella del campo la tabella si disallinea

Il manipolatore `setw()` setta la lunghezza del campo. `setfill()` setta il carattere di riempimento (per default lo spazio). `left` e `right` impongono l'allineamento a sinistra e a destra (default) del testo all'interno del campo.

# Scrittura su file
# Lettura da file

## File di testo vs. file binari

- File binari (binary files):

```
0C 22 38 4E
```

(cifre esadecimali)

- Il contenuto sono *byte* senza una interpretazione a priori
- Il significato di tali byte lo decide di volta in volta il programmatore
- Vantaggi:
  - più flessibile
  - più compatti
  - l'interpretazione non dipende dalla piattaforma

- File di testo (text files):

```
····12··34
56····78
```

- Il contenuto sono byte interpretati come *testo ASCII*
- Il significato di tale testo lo decide di volta in volta il programmatore
- Piattaforme diverse hanno formati di testo diversi
- Vantaggi:
  - Possono essere visualizzati e modificati con un text editor (notepad, gedit, etc.)

Nell'informatica esistono due tipi di file: i file binari ed i file di testo. I file binari contengono *byte "grezzi"*, cioè senza una interpretazione predefinita, il cui significato viene deciso dal programmatore. I file di testo invece vengono interpretati come *testo ASCII*, il cui significato viene deciso dal programmatore. Il sistema operativo non si accorge di questa differenza. È il programmatore che sceglie di aprire un file in modalità "binaria" o "di testo".

Memorizzare dei dati un forma binaria è più flessibile e più efficiente dal punto di vista dello spazio di memorizzazione. D'altra parte, i file di testo hanno il grande vantaggio di essere visualizzabili e modificabili con un qualsiasi editor di testo, per esempio notepad o gedit.

Notare che un file di testo può essere aperto in modalità binaria. In questo caso, la libreria iostream non cercherà di interpretare i byte come codici ASCII. Al contrario, è sconsigliato aprire un file binario in modalità testo.

ATTENZIONE! Per convenzione, nei file di testo su piattaforme Microsoft gli "accapi" vengono codificati con due byte (carriage return + line feed), mentre in quelle non-Microsoft vengono codificati con un solo byte (line feed). Se aprite il file in modalità testo, la libreria iostream leggerà e scriverà gli accapi in modo diverso a seconda della piattaforma, quindi non dovrete preoccuparvi di questo aspetto.

Solitamente, i file più complessi e strutturati vengono memorizzati in modalità binaria. I file di testo vengono usati solo per i semplici file di configurazione dei programmi. Con l'avvento di XML i file di testo stanno ritornando di moda anche per i formati complessi. Questo perché con XML è possibile definire formati complessi e strutturati che sono anche visualizzabili tramite editor.

## Esempio – Scrittura su file di testo

```
1   #include <iostream>
2   #include <fstream>
3   using namespace std;
4   int main(){
5         int i = 120;
6         ofstream f("out.txt", ios::out);
7         f << "Hello, world!" << endl;
8         f << "i = " << i << endl;
9         f.close();
10  }
```

In C++, i file vengono gestiti con degli speciali stream chiamati *file stream*. Ad ogni file stream è associato (in lettura o scrittura) un file.

Questo programma scrive su un file di testo mediante uno stream C++. Lo stream `f` rappresenta il file. Al momento della definizione bisogna specificare il nome del file e la modalità di apertura (per esempio `ios::out`).

Una volta creato il file stream, si possono utilizzare tutti i metodi degli stream. Sul file vengono scritti i messaggi "`Hello, world!`" (accapo) "`i = 120`" (accapo). Infine, il file viene chiuso. Questo equivale a spedire la marca di fine stream.

Per usare i file stream si deve includere la libreria `<fstream>` (senza .h).

ATTENZIONE! Anche i file stream sono bufferizzati. Scrivere sul buffer del file stream non implica quindi l'effettiva scrittura sul file. La scrittura effettiva avviene quando si viene scritto sul buffer un accapo, quando si usa il manipolatore flush, quando si invoca il metodo close(), oppure nel distruttore del file stream.

## Esempio – Lettura da file di testo

```
1   #include <iostream>
2   #include <fstream>
3   using namespace std;
4   int main(){
5         int i;
6         ifstream f("in.txt", ios::in);
7         f >> i;
8         cout << "i = " << i << endl;
9         f.close();
10  }
```

Questo programma legge un intero (in caratteri ASCII) da un file di testo e lo stampa a video. Successivamente chiude il file. Per semplicità, nell'esempio sopra non vengono gestiti gli errori.

## Operazioni sui file

- Modalità di apertura di un file (flag da combinare con l'or bit-a-bit):
  - Modalità base:
    - `ios::in` → lettura
    - `ios::out` → scrittura (se il file non esiste viene creato; se il file esiste viene cancellato e ricreato)
    - `ios::app` → aggiunta (se il file non esiste viene creato; se il file esiste non viene cancellato, ma si aggiungono caratteri alla fine)
  - Modalità aggiuntive (si combinano con l'or bit-a-bit: `|`):
    - Aggiungere `|ios::binary` → per lettura/scrittura/aggiunta su file binari
- Il metodo `close()` provvede a scaricare il buffer e chiudere il file associato allo stream. È invocato automaticamente dal distruttore.
- `<fstream>` non permette l'*accesso esclusivo* ai file.
  - Mentre sto leggendo/scrivendo un file qualche altro programma potrebbe leggerci/scriverci.
  - L'accesso esclusivo è ottenibile tramite librerie *platform-specific*.

Nella classe `ios` sono definiti, tramite enumerati, varie modalità di apertura di un file. La modalità `ios::in` indica la modalità di *lettura*. Se il file non esiste viene prodotto un errore. La modalità `ios::out` indica la *sovrascrittura*. Se il file non esiste viene creato vuoto. Se il file esiste ne viene cancellato il contenuto. La modalità `ios::app` indica l'*aggiunta* (*append*). Se il file non esiste viene creato vuoto. Se il file esiste, i dati vengono aggiunti alla fine del vecchio contenuto.

A queste tre modalità di base possono essere specificate altre flag (tramite or bit-a-bit). `ios::binary` specifica che il formato dello stream è binario, gli operatori put to e get from spediscono e ricevono i dati direttamente in formato binario, e non in cifre. `ios::nocreate` fa fallire l'apertura se il file non esiste (utile per la modalità `ios::app`). `ios::noreplace` fa fallire l'apertura se il file esiste (utile per la modalità `ios::out`).

Il metodo `close()` è invocato automaticamente dal distruttore. Tuttavia è una buona pratica di programmazione esplicitare la chiamata a `close()`, per rendere più leggibile il codice.

È importante sapere che la libreria `<fstream>` non permette l'*accesso esclusivo* ai file. Questo vuol dire che mentre accedo in scrittura o lettura ad un file, qualcun altro processo può accedervi in scrittura o lettura, creando così delle *corse critiche*. Se si vuole evitare questo comportamento bisogna ottenere l'accesso esclusivo al file tramite metodi di librerie *platform-specific* (cioè funzionanti solo su Windows o solo su Linux, etc.). Ovviamente questo riduce la *portabilità* del mio codice.

## Lettura/scrittura da file binari

- Lettura
  - Operatore >>

```
1  int a;
2  infile >> a;
```

Nessuna conversione ASCII!

  - `read()`

```
1  unsigned char buf[100];
2  int letti = infile.read(buf, 100);
```

- Scrittura
  - Operatore <<

```
1  int b = 500
2  outfile << b;
```

Nessuna conversione ASCII!

  - `write()`

```
1  unsigned char buf[100];
2  outfile.write(buf, 100);
```

Per leggere o scrivere su file binari, si possono utilizzare gli operatori `>>` e `<<` ed i metodi `read()` e `write()`.

In particolare, `>>` e `<<` leggono e scrivono una singola variabile (per esempio un intero) senza fare conversioni ASCII. La variabile viene quindi letta o scritta nella sua rappresentazione binaria, e in lettura non vengono saltati gli spazi iniziali.

`read(buf, 100)` legge 100 byte (senza interpretarli) dal buffer interno del file stream e li copia nel buffer utente `buf`. Restituisce la quantità di byte effettivamente letta, che può essere minore di 100 nel caso in cui il file sia terminato.

`write(buf, 100)` scrive 100 byte (senza interpretarli) dal buffer utente `buf` al buffer interno del file stream.

## Esempio – Copia esatta di un file

```
1   #include <fstream>
2   using namespace std;
3   int main(){
4         ifstream ingr("source.txt", ios::in|ios::binary);
5         ofstream usc("dest.txt", ios::out|ios::binary);
6         if (ingr.fail() || usc.fail()) exit(-1);
7         char buf[128];
8         int letti = ingr.read(buf, 128);
9         if (ingr.bad()) exit(-1);
10        while (!ingr.eof()){
11              usc.write(buf, letti); if (usc.bad()) exit(-1);
12              letti = ingr.read(buf, 128);
13              if (ingr.bad()) exit(-1);
14        }
15        usc.write(buf, letti); if(usc.bad()) exit(-1);
16        ingr.close();
17        usc.close();
18  }
```

Questo programma copia il contenuto di un file su un altro file. Si apre in lettura il file “source.txt” e in scrittura “dest.txt”. I metodi `open()` sono un alternativa ad usare il costruttore con argomenti. Viene fatto il controllo degli errori (se il file di input non esiste o non può essere letto, se il file di output non può essere scritto). Si legge dal file di ingresso 128 byte per volta (`read()`) e si scrive in uscita (`write()`) fino alla fine del file in ingresso. Infine si chiudono entrambi i file. Notare che nell'esempio, dei file di testo (estensione .txt) vengono aperti in modalità binaria.

La riga 6 controlla che non ci siano stati errori nell'apertura dei file. In particolare “source.txt” potrebbe non esistere e “dest.txt” potrebbe non essere scrivibile (per esempio un file protetto o già aperto da un altro programma).

Le righe 11 e 15 controllano che non ci siano stati errori in lettura. Le righe 13 e 17 controllano che non ci siano stati errori in scrittura. Ovviamente è preferibile centralizzare la gestione degli errori con la tecnica delle eccezioni.

## Gestione del cursore

- Negli stream che le supportano (per esempio i file) sono disponibili metodi per riposizionare il cursore di input:
  - `long pos = infile.tellg();`
  - `infile.seekg(pos);`

  e quello di output:
  - `long pos = outfile.tellp();`
  - `outfile.seekp(pos);`

Supponiamo si voglia iniziare a leggere da metà file senza leggere tutta la parte iniziale. Per far questo sono previsti dei metodi per riposizionare i *cursori* di lettura e di scrittura. La posizione del cursore indica il prossimo carattere che verrà letto/scritto. Si distingue tra cursore di lettura e di scrittura perché negli stream bidirezionali, per esempio `fstream`, è possibile leggere una zona del file e contemporaneamente scrivere su un'altra zona.

`tellg()` e `seekg()` rispettivamente restituiscono e assegnano la posizione del cursore di lettura. `tellp()` e `seekp()` sono gli analoghi per il cursore di scrittura. Ovviamente questi metodi hanno effetto solo per quegli stream in cui ha senso il concetto di cursore, per esempio i file stream. Non hanno nessun effetto per `cin` e `cout`.

## Gestione centralizzata degli errori

```
1   #include <iostream>
2   using namespace std;
3   int main(){
4         ifstream infile("source.txt", ios::in);
5         // ordino di sollevare eccezioni con failbit o eofbit:
6         infile.exceptions(ios::failbit | ios::eofbit);
7         try{
8               infile >> ...;
9               infile >> ...;
10              infile.getline(..., 100);
11        }
12        catch (ios_base::failure){
13              if (infile.eof())
14                    cerr << "Il file è finito" << endl;
15              else
16                    cerr << "Il file non ha il formato giusto" << endl;
17        }
18        infile.close();  }
```

Per separare l'algoritmo dalla gestione degli errori, si può dire ad uno stream di lanciare *eccezioni* in caso di errori. Il meccanismo è quello standard delle eccezioni C++. Il metodo `exceptions()` dice allo stream i casi in cui deve sollevare un'eccezione.

Questo programma legge cinque interi da tastiera. In caso di errore di formattazione o fine dello stream stampa dei messaggi di errore.

# Stringhe C++

## Il tipo di dati `std::string`

- Il tipo di dati `std::string` è disponibile nella C++ Standard Library. È necessario includere il file di intestazione `<string>`
  - `<string>` non deve essere confuso con `<string.h>` (o `<cstring>`), che raccoglie le funzioni standard per il trattamento di stringhe C.
- Una `string` può non essere implementata tramite un semplice vettore. Di solito si usano delle strutture dati ottimizzate per le operazioni più comuni sulle stringhe (aggiunta, inserimento di sottostringhe, ricerca di sottostringhe, etc.). Per esempio liste di vettori dinamici.
- L'allocazione di una `string` è *gestita* (*managed*). Una `string` viene allocata, riallocata e deallocata automaticamente quando necessario, senza necessità di intervento da parte del programmatore.

Nel C di fatto non esiste il tipo stringa. Si parla impropriamente di “stringhe C” riferendosi ai puntatori a carattere, ma questi hanno un comportamento radicalmente diverso dai normali tipi predefiniti, come l'`int` o il `double`. In particolare gli operatori di assegnamento (`=`) e confronto (`!=`) hanno un significato diverso da quello che ci si aspetterebbe. Inoltre si deve continuamente fare attenzione alla dimensione della memoria allocata.

In C++ invece viene definito un tipo stringa a tutti gli effetti: `std::string`. Per usarlo bisogna includere `<string>` (senza .h). Il tipo `string` di solito non viene realizzato tramite un array ma con strutture più complesse, ottimizzate per l'inserimento e l'estrazione di sottostringhe, per esempio liste di array. Agli effetti esterni, però, una `string` non è altro che una sequenza di `char`. Il programmatore non deve preoccuparsi di gestirne l'allocazione di memoria. Il tipo `string` rialloca automaticamente la memoria quando la lunghezza della stringa aumenta.

# Consiglio:

Non usate *mai più* `char*` per elaborare stringhe
Usate sempre `string`

Eseguire manipolazioni complesse di stringhe C è una pratica sconsigliata, in quanto: (1) è soggetta a errori di programmazione (*error prone*) a causa della complessa matematica dei puntatori; (2) gli errori introdotti sono difficilmente identificabili e causano guasti difficilmente ripetibili; (3) il programma risulta poco leggibile e poco intuitivo.

## Creazione di oggetti `string`

- `string s1;` → stringa vuota
- `string s2 = "pippo";` → conversione da stringa C
- `string s3 = s4;` → copia per valore da stringa C++

Per creare un oggetto `string` posso crearlo vuoto, convertirlo da una stringa C, oppure copiarlo (per valore) da un altro oggetto `string`.

## Operazioni comuni

- `str1 = str2;` → Copia per valore
- `str3 = str1 + str2;` → Concatenazione
- `str2 += str1;` → Concatenazione e copia per valore
- `if(str1 == str2)` → Confronto per valore
- `str1[10] = 'b';` → Accesso (lettura o scrittura) al carattere di indice 10
- `int len = str1.length();` → Lunghezza
- Con =, +, +=, ==, !=, uno degli operandi può essere una costante letterale o una stringa C:
  - `str1 = "abc";`
  - `str1 = str2 + "abc";`
  - `str1 = "abc" + str2;`
  - `if(str1 == "abc") {...}`
  - etc...

Prima le stringhe C vengono convertite in `string`, poi vengono invocati gli operatori

Il tipo `string` ridefinisce alcuni operatori. L'operatore `=` esegue una copia per valore, l'operatore `+` una concatenazione tra stringhe. L'operatore `+=` esegue una concatenazione ed una copia per valore in modo ottimizzato. L'operatore `==` restituisce true se due stringhe sono uguali, cioè se hanno la stessa lunghezza e contengono la stessa sequenza di caratteri. L'operatore `!=` restituisce true se due stringhe sono diverse. Notare che questi operatori ridefiniti di `string` assumono un significato completamente diverso rispetto a quelli delle stringhe C.

Con l'operatore ridefinito `[]` si accede ad un carattere della stringa, in modo analogo a quanto succede per le stringhe C.

Per recuperare la lunghezza della stringa si usa il metodo interno `length()`.

Negli operatori ridefiniti `=`, `+`, `==` e `!=`, uno degli operandi può essere una costante letterale o una stringa C. In questo caso la costante letterale o stringa C viene prima convertita in `string`, e poi viene invocato l'operatore.

## Esempio – Acquisizione e stampa di tre parole

```
1   #include <string>
2   #include <iostream>
3   using namespace std;
4   int main(){
5       cout << "Inserisci tre parole." << endl;
6       string s1;
7       string s2;
8       string s3;
9       cin >> s1 >> s2 >> s3;
10      string s4 = "Parole inserite: " + s1 + ", " + s2 + ", " + s3 + "\n";
11      cout << s4;
12  }
```

Output a video:

```
Inserisci tre parole.
pippo      pluto      topolino
Parole inserite: pippo, pluto, topolino
```

Questo programma chiede all'utente di inserire tre parole da tastiera e poi le stampa su schermo. Notare che l'operatore `>>` degli input stream applicato a una `string` legge una parola, analogamente a quanto succede per le stringhe C.

## C vs. C++

- Allocazione, assegnamento, concatenazione, confronto:

Vecchio C

```c
#include <string.h>
int main()
{
    char s1[] = "abc";
    char s2[4];
    strcpy(s2, s1);
    strcat(s2, "def");
    if(strcmp(s2, s1) == 0)
        // ...
}
```

Attenzione alla capacità del buffer!

```c
#include <string>
int main()
{
    string s1 = "abc";
    string s2;
    s2 = s1;
    s2 += "def";
    if(s2 == s1)
        // ...
}
```

- Evitare il buffer overflow:

errore classico

```c
char s1[100];   char s2[100];
cin >> s1; // una parola
cin.getline(s2); // una riga
```

```c
string s1;   string s2;
cin >> s1; // una parola
getline(cin, s2); // una riga
```

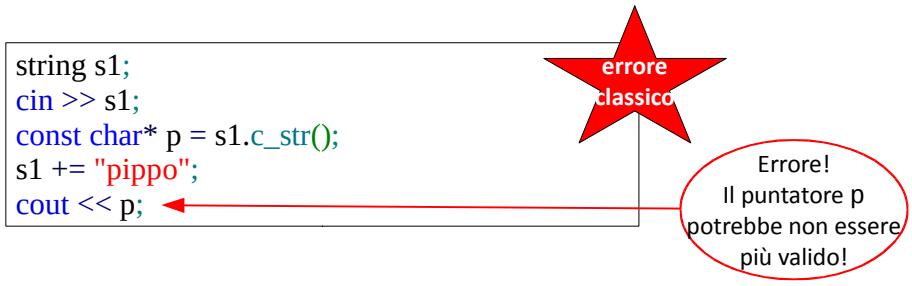
Con i tipi string non si usano più le operazioni C `strcpy()`, `strcat()`, `strcmp()`, etc. ma si fa tutto tramite operatori. Il codice risulta più intuitivo e leggibile. L'assegnamento (=) corrisponde al vecchio `strcpy()`, la concatenazione e assegnamento (+=) corrisponde al vecchio `strcat()`, e l'uguaglianza (==) allo `strcmp()`.

Notare che nell'eseguire l'operatore get from, stavolta non dobbiamo più preoccuparci di aver allocato abbastanza memoria. Il tipo `string` alloca automaticamente la memoria necessaria.

Anche per le stringhe C++, l'operatore get from fa l'input di singole *parole*. Per acquisire righe intere (fino all'accapo) bisogna usare il metodo `getline()`, definito stavolta come globale.

## Conversione tra stringhe C e stringhe C++

- Da `char*` a `string`:
  - `char cstr1[] = "pippo"; (string)cstr1;` → (cast)
  - `str1 = "pippo";` → (assegnamento)
- Da `string` a `char*`:
  - `str1.c_str();` → (metodo `c_str()`)




Per convertire da stringa C a stringa C++ si può usare il cast o un semplice assegnamento. Per convertire da stringa C++ a stringa C si usa il metodo `c_str()` che restituisce un puntatore ad una stringa C *costante* che ha lo stesso valore della stringa C++.

ATTENZIONE! La stringa C viene memorizzata in uno spazio di memoria temporaneo interno all'oggetto `string`. Se si memorizza il puntatore `c_str()` e poi si modifica la `string`, il puntatore viene invalidato. L'esempio sopra si corregge sostituendo `cout << s1.c_str()` al posto dell'ultima riga.

## Gestione del buffer interno

- Il contenuto di una `string` è memorizzato in un buffer interno con una certa capacità
- Se la lunghezza della `string` supera tale capacità, il buffer viene automaticamente riallocato e il contenuto copiato nel nuovo buffer
- `int cap = str1.capacity();` → Capacità del buffer interno
- `str1.reserve(100);` → Impone un buffer di capacità almeno 100 (può provocare una riallocazione)

```
1   #include <string>
2   #include <iostream>
3   using namespace std;
4   int main(){
5       string s1;
6       s1.reserve(20);
7       s1 += "1234567890";
8       s1 += "abcdefghij";
9       cout << s1;
10  }
```

Il metodo `capacity()` restituisce la grandezza della memoria interna allocata per la stringa (sempre maggiore o uguale alla lunghezza della stringa). Il metodo `reserve()` serve a forzare la libreria ad allocare una certa quantità di memoria interna. È utile quando si costruisce una stringa per concatenazioni successive, ma si sa già quanto sarà la lunghezza finale. Usare `reserve()` permette di evitare le continue riallocazioni automatiche, aumentando così l'efficienza complessiva.

## Sottostringhe

- Sono definite le operazioni di inserimento, cancellazione e sostituzione di sottostringhe.
  - `s1.insert(pos, s2);`
  - `s1.erase(pos, len);`
  - `s1.replace(pos, len, s2);`
- È definita l'operazione di recupero di una sottostringa (copia per valore).
  - `s2 = s1.substr(pos, len);`

Sono definiti gli operatori di concatenazione (+) e concatenazione e assegnamento (+=). Funzionano sia tra due `string` che tra una `string` e una stringa C.

Si possono inserire, cancellare, sostituire e recuperare sottostringhe. `insert()` inserisce la sottostringa `s2` nella posizione `pos`, spostando tutti gli altri caratteri a destra. `erase()` cancella `len` caratteri a partire dalla posizione `pos`. `replace()` sostituisce `len` caratteri a partire dalla posizione `pos` con la stringa `s2`, che può essere di lunghezza diversa da `len`. Equivale ad una chiamata ad `erase()` più una chiamata ad `insert()` ma è più veloce.

`substr()` copia in `s2` la sottostringa di `s1` che parte dalla posizione `pos` ed è lunga `len` caratteri.

## Confronti lessicografici e ricerche di sottostringhe

- Sono definiti gli operatori di confronto. Utilizzano l'ordinamento lessicografico.
  - `bool operator!=(const string&, const string&)`
  - `bool operator==(const string&, const string&)`
  - `bool operator<(const string&, const string&)`
  - …
- Ci sono anche metodi per la ricerca di caratteri e sottostringhe.
  - `int i = s1.find("cde", pos);` → Cerca, nelle posizioni maggiori o uguali di `pos`, la prima occorrenza di "`cde`" all'interno di `s1`. Se la trova, ne restituisce la posizione, altrimenti restituisce `string::npos`. Il valore di default di `pos` è `0`.
  - `int i = s1.find_first_of("cde", pos);` → Cerca, nelle posizioni maggiori o uguali di `pos`, la prima occorrenza di '`c`', di '`d`' o di '`e`' all'interno di `s1`.
  - `int i = s1.rfind(s2, pos);` → Stessa cosa di `find` ma cerca l'*ultima* occorrenza nelle posizioni *minori* di `pos`. Il valore di default di `pos` è `string::npos`.
  - `int i = s1.find_last_of("cde", pos);` → Stessa cosa di `find_first_of` ma cerca l'*ultima* occorrenza nelle posizioni *minori* di `pos`.

Sono definiti gli operatori di confronto per stabilire se due stringhe sono uguali (==) o diverse (!=) o se una viene prima dell'altra (<) nell'ordine lessicografico. Le maiuscole sono considerate "minori" di tutte le minuscole, quindi `"Pippo" < "pippo"`. Sono definiti tutte le combinazioni di operatori di confronto (<, >, <=, >=).

Sono definiti dei metodi di ricerca. `find()` cerca nelle posizioni maggiori o uguali a `pos` (default: 0) la prima occorrenza di una sottostringa. Se la trova ne restituisce la posizione, altrimenti restituisce `string::npos`, che indica una posizione non valida. `rfind()` fa la stessa cosa ma cercando a partire dalla posizione precedente a `pos` all'indietro. Il valore di default di `pos` è `string::npos`, che corrisponde a cercare dall'ultima posizione. `find_first_of()` fa la stessa cosa di `find()` ma cerca uno dei caratteri della sottostringa (e non la sottostringa completa). `find_last_of()` è il corrispettivo che cerca dalla posizione precedente a `pos` all'indietro.

## Esempio – Cercare tutte le occorrenze

```
1  #include <string>
2  #include <iostream>
3  using namespace std;
4  int main(){
5        string s1;
6        cin >> s1;
7        int pos = 0;
8        for(;;){
9              pos = s1.find("cd", pos);
10             if (pos == string::npos) break;
11             cout << "pos = " << pos << endl;
12             pos += 2;
13       }
14 }
```

```
abcdcdabcd
pos = 2
pos = 4
pos = 8
```

Questo programma prende una parola da tastiera e cerca tutte le occorrenze della sottostringa "`cd`". Per fare questo invoca ripetutamente il metodo `find()` partendo allinizio da `pos=0`, e poi ogni volta dalla posizione successiva all'ultima occorrenza trovata.

## Scambio di due stringhe

- Versione inefficiente (esegue tre copie per valore):

```
string app = s1;
s1 = s2;
s2 = app;
```

- Versione efficiente (scambia solo i puntatori interni):

```
swap(s1, s2);
```

Per scambiare due stringhe, la STL fornisce il metodo globale `swap()`, che permette di evitare di eseguire tre assegnamenti con una stringa di appoggio. Oltre ad essere più leggibile, il metodo `swap()` è anche più efficiente in quanto scambia i puntatori interni anziché scambiare il contenuto delle stringhe.

## Codificare un intero in ASCII

```
1   #include <string>
2   #include <iostream>
3   using namespace std;
4   void funz1(const string& str){...}
5   int main(){
6         cout << "Inserisci un intero." << endl;
7         int intero;
8         cin >> intero;
9         funz1(intero);
10  }
```

**Errore!**
**Manca conversione ASCII**

Un problema comune è quello di codificare interi (o double) in una stringa in codifica ASCII e viceversa. L'esempio sopra produce errore in quanto la funzione `funz1` vuole l'intero nella sua rappresentazione ASCII.

## istringstream e ostringstream

- Rappresentano stream da e verso delle stringhe (*string stream*)
- Supportano gli operatori >> e << che eseguono conversioni ASCII
- Il metodo interno `str()` restituisce la stringa risultante sotto forma di `string`

```
1   #include <string>
2   #include <iostream>
3   using namespace std;
4   void funz1(const string& str){...}
5   int main(){
6         cout << "Inserisci un intero." << endl;
7         int intero;
8         cin >> intero;
9         ostringstream oss;
10        oss << intero;
11        funz1(oss.str());
12  }
```

Per fare questo, in C++ si usano gli *string stream*, che sono speciali stream che leggono o scrivono in un oggetto di tipo `string`. Come tutti gli stream, esportano gli operatori `>>` e `<<` che eseguono conversioni da e verso codifica ASCII. Per trasformare un intero in codifica ASCII si usa uno string stream di output, si immettono gli interi che si vogliono convertire e poi si accede alla stringa risultante con il metodo `str()`.

## Decodificare ASCII in un intero

```
1   #include <string>
2   #include <iostream>
3   #include <sstream>
4   using namespace std;
5   string funz2(){...}
6   int main(){
7         string stringa = funz2();
8         int intero;
9         istringstream iss(stringa);
10        iss >> intero;
11        if (iss.fail()) cerr << "Errore conversione ASCII!" << endl;
12        else cout << "L'intero vale: " << intero << endl;
13  }
```

Al contrario, per trasformare una rappresentazione ASCII in un intero si usa uno string stream in input che viene associato ad un oggetto `string` al momento della creazione. Poi si usa l'operatore `>>` che funziona come di consueto, e che produce `fail()` se la stringa non ha il formato giusto (per esempio contiene lettere anziché cifre).

## Confronto con C/C++

- In C si usavano le funzioni `itoa` (integer to ASCII) e `atoi` (ASCII to integer).

Vecchio C

```
#include <cstdlib.h>
int main()
{
      int intero;   char buf[5];
      cin >> intero;
      itoa(intero, buf, 10);
      funz1(buf);
}
```

```
#include <sstream>
int main()
{
      int intero;   ostringstream oss;
      cin >> intero;
      oss << intero;
      funz1(oss.str());
}
```

Attenzione alla capacità del b

Vecchio C

```
#include <cstdlib.h>
int main()
{
      char buf[10];
      cin >> buf;
      funz1(atoi(buf));
}
```

Se il buffer non contiene cifre?

```
#include <sstream>
int main()
{
      string stringa;
      cin >> str;
      istringstream iss(stringa);
      iss >> intero; if(iss.fail()) {...}
      funz1(intero);   }
```

Nella libreria standard del C, per convertire da e verso codifica ASCII si usavano le funzioni globali `atoi` (ASCII to integer) e `itoa` (integer to ASCII). `itoa` esponeva ai consueti rischi di buffer overflow. Mentre `atoi` aveva dei problemi riguardo alla gestione degli errori di decodifica.

## Conversioni non in base 10

- Si utilizzano i manipolatori.
- Base 8:

```
iss >> oct >> intero;
oss << oct << intero;
```

- Base 16:

```
iss >> hex >> intero;
oss << hex << intero;
```

- Per riportare in base 10:

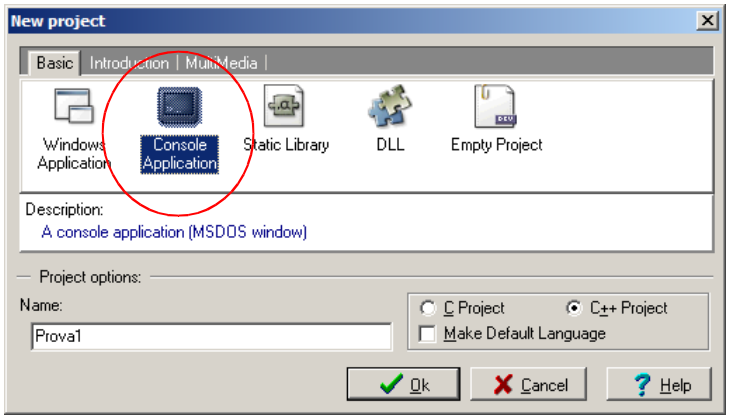
```
iss >> dec;
oss >> dec;
```

Per effettuare conversioni da e verso ASCII in basi diverse da 10 si usano i consueti manipolatori: `oct` per la base ottale, `hex` per la base esadecimale e `dec` per riportare in base 10.

# Esercizio riassuntivo

# L'ambiente Dev-C++

- Start → Programmi → Bloodshed Dev-C++ → Dev-C++
- File → New → Project...




- Salvare il project file in una cartella.
- Ricordarsi sempre il `system("PAUSE")` alla fine del programma.

# Esercizio riassuntivo

- Realizzare un programma che:
  - prende da tastiera 5 parole (stringhe C++);
  - le ordina;
  - le stampa su un file “strings.txt” nel seguente formato:

```
minnie
pippo*
pluto*
topolino
zapotec*
```

Dove “*” indica che la parola ha un numero dispari di caratteri.

# Esercizio riassuntivo – Soluzione

```
1   #include <fstream>
2   #include <iostream>
3   #include <string>
4   using namespace std;
5   void bubbleSort(string a[5]){
6       for (int i = 0; i < 4; i ++){
7           for (int j = i+1; j < 5; j++){
8               if (a[j-1] > a[j])
9                   swap(a[j-1], a[j]);
10          }
11      }
12  }
13  int main(){
14      ofstream usc;
15      string words[5];
16      int i;
```

# Esercizio riassuntivo – Soluzione

```
17      // leggo 5 parole da tastiera:
18      cin >> words [0];
19      i = 1;
20      while (!cin.fail() && i < 5){
21          cin >> words[i];
22          i++;
23      }
24      if(cin.fail()){
25          cerr << "Error in getting input" << endl;
26          exit(-1);
27      }
28      // ordino nell'ordine lessicografico:
29      bubbleSort(words);
30      // stampo nel file:
31      usc.open("strings.txt", ios::out);
32      if(usc.fail()) exit(-1);
```

# Esercizio riassuntivo – Soluzione

```
33          for(i = 0; i < 5; i++){
34                  usc << words[i];
35                  if(words[i].length()%2 == 1)
36                          usc << '*';
37                  usc << endl;
38          }
39          usc.close();
40  }
```

# Alcuni riferimenti

- Bjarne Stroustroup, The C++ Programming Language, Addison-Wesley
- Daniela Dorbolo, Graziano Frosini, Beatrice Lazzerini, Programmazione a oggetti con riferimento al C++, Franco Angeli, 2000, capitolo 10
- http://www.parashift.com/c++-faq-lite/index.html, C++ FAQ Lite, Frequently Asked Questions
- http://www.bo.cnr.it/corsi-di-informatica/corsoCstandard/Lezioni/38IOCPP.html Le operazioni di input-ouput in C++ (Dr. Antonino Ficarra, Dr. Matteo Murgia)
- iostream reference guide http://www.cplusplus.com/ref/iostream/