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

Pericle Perazzo

pericle.perazzo@iet.unipi.it

http://info.iet.unipi.it/~fondii/

3 marzo 2012

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 `<<` (get from) simboleggia la direzione dei dati che dalle variabili vanno al video.

Come si vede nell'esempio, le chiamate a get from possono essere concatenate su una sola riga di codice. Questo perché ogni chiamata a get from restituisce il riferimento a `cout` per la prossima chiamata. `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 get from: il “put to” (>>). Anche questo simboleggia la direzione dei dati che dalla tastiera raggiungono le variabili.

Vantaggi di `cout` e `cin`

- Sono *type-safe*:

```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.

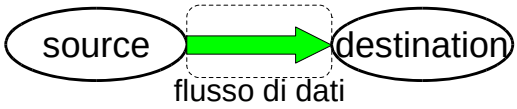
Prima di tutto, `cin` e `cout` sono type-safe. Con `printf` siamo obbligati a dichiarare il tipo della variabile da stampare (`"%s"`) e poi il nome della variabile (`a`). Cosa succede se il tipo della variabile non coincide con il tipo dichiarato? Il compilatore non dà errore ma il mio programma avrà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

Un altro vantaggio è la possibilità di estendere gli operatori put to e get from 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).

6

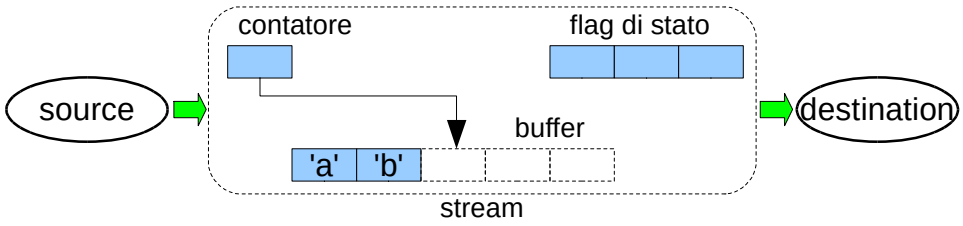
Uno *stream* (flusso) è un oggetto che astrae una semplice comunicazione da una sorgente ad una destinazione. Uno dei due estremi (sorgente o destinazione) è il programma che sto scrivendo. All'altro estremo può esserci un dispositivo fisico, un file, una connessione di rete, un altro programma, etc. 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 sempre di uno stesso tipo. Noi tratteremo solo stream di *caratteri*.

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

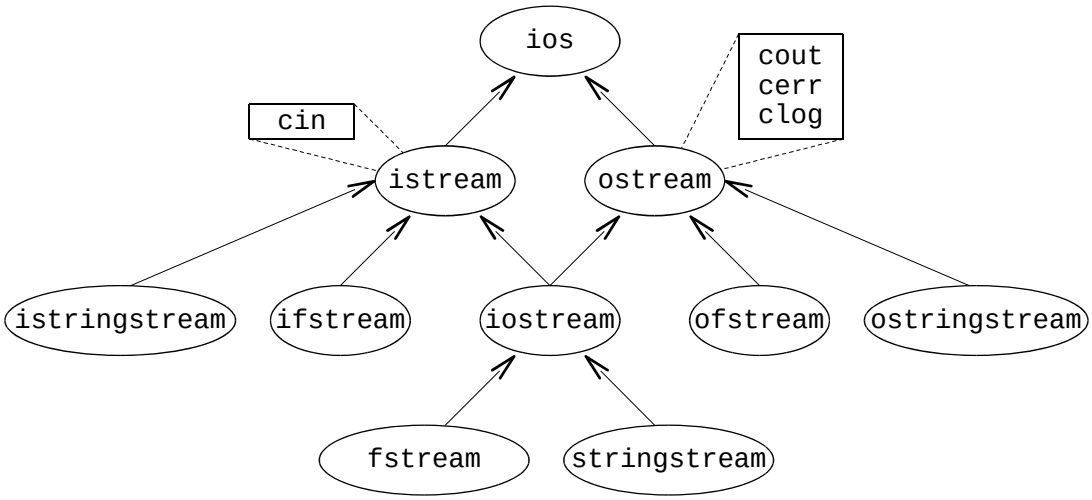
- Un flusso è implementato con un *buffer* e delle strutture dati di appoggio (contatori, indicatori di stato, etc.)



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.

All'interno dello stream ci sono anche delle flag di stato che indicano l'esito (successo/errore) dell'ultima operazione.

La gerarchia `std::ios`



Questa è la gerarchia (molto semplificata) della libreria iostream, che implementa gli stream in C++. La classe `ios` racchiude le funzionalità comuni a tutti gli stream. `istream` è un generico stream di input. `ostream` è un generico stream di output. l'oggetto `cin` è un'istanza di `istream`, mentre l'oggetto `cout` è un'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 uno 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 memorizzata.

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 f(...);
10      myfun(f, 3, 4);
11      // scrivo su stringa memorizzata:
12      ostringstream s(...);
13      myfun(s, 5, 6);
14  }
```


Come accennato sopra, il terzo vantaggio di `cout` e `cin` è che possono essere usati come *stream generici*. In questo esempio, la funzione `myfun()` invia dei byte ad uno stream generico 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`

- Tutte le classi sono specializzazioni di *template* (il parametro di template è il tipo di ogni singola "cella" dello stream). Normalmente avremo a che fare con *stream di caratteri*.
- `std::ios` ingloba alcune funzionalità comuni a tutti i tipi di stream:
  - buffer,
  - controllo degli errori (flags)
  - formati di apertura/lettura/scrittura dello stream.
- In particolare i metodi `good()`, `fail()`, `bad()`, `eof()` permettono di capire se l'ultima operazione effettuata sullo stream rispettivamente è andata a buon fine, ha riportato un errore, ha riportato un errore irrecuperabile, o si è raggiunto la marca di fine stream.
- Il metodo `clear()` "resetta" la condizione di errore (se recuperabile) e sblocca lo stream.

Tutte le classi sono in realtà specializzazioni di template, con parametro il tipo del dato (per noi `char`). `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 irrecuparabile. 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

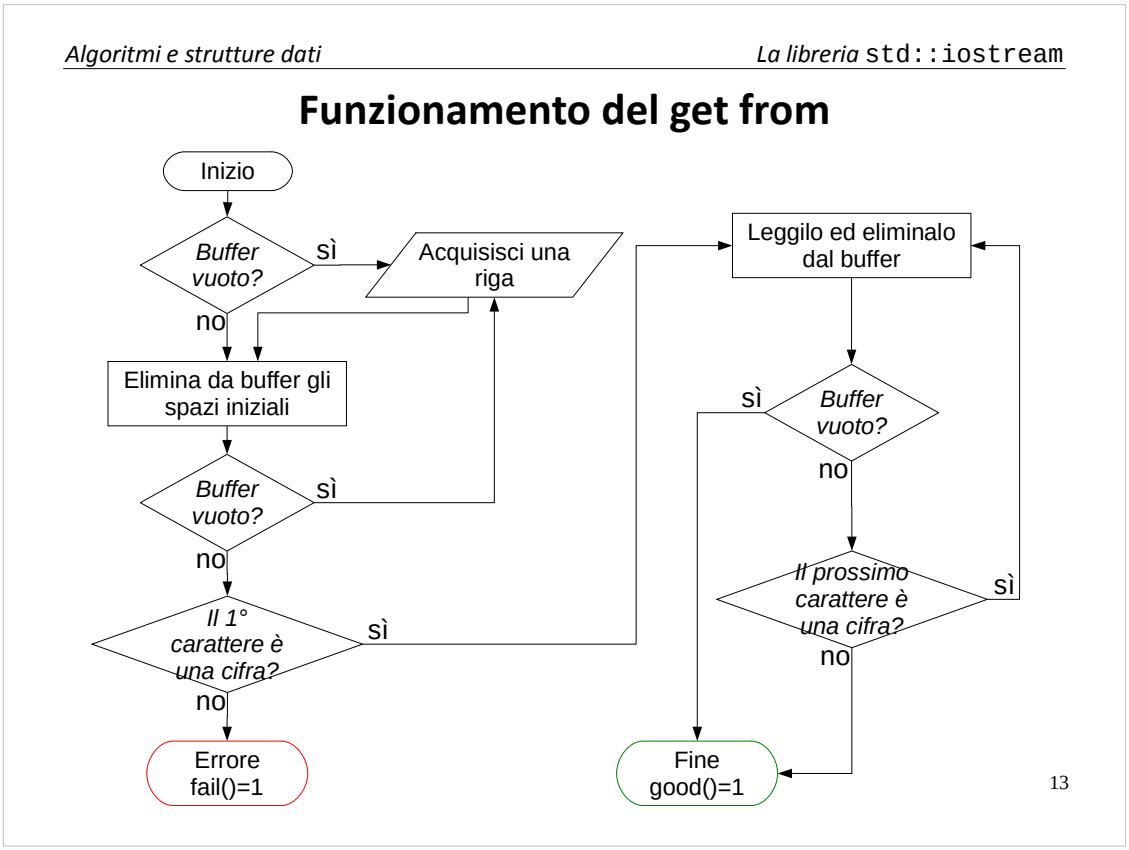
- Contiene metodi e campi per l'*output* su stream
- `std::cout`, `std::cerr`, `std::clog` sono istanze di `ostream`
- Per i tipi predefiniti e già ridefinito l'operatore *put to* (<<):
  - `int i = 1; std::cout << i;`
  - `double d = 3.14; std::cout << d;`
  - `char c = 'z'; std::cout << c;`
  - `char s[] = "pippo"; std::cout << s;`
- Per i tipi utente occorre definire funzioni globali (eventualmente `friend` se vogliamo che accedano a campi privati):
  - `ostream& operator<<(ostream& out, const MyClass& m){...}`
    `MyClass m; std::cout << m;`
- Da `std::ostream` derivano:
  - `ofstream` per scrivere su file
  - `ostringstream` per scrivere su una stringa in memoria. Ogni operazione di scrittura ingrandisce opportunamente la stringa. Utile per preparare un output formattato.

`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 e campi per l'*input* da stream
- `std::cin` ne è un istanza
- Per i tipi predefiniti e già ridefinito l'operatore *get from* (>>):
  - `int i; std::cin >> i;`
  - `double d; std::cin >> d;`
  - `char c; std::cin >> c;`
  - `char s[1000]; std::cin >> s;`
- Per i tipi utente occorre definire funzioni globali (eventualmente `friend` se vogliamo che accedano a campi privati):
  - `istream& operator>>(istream& in, MyClass& m);`
    `MyClass m; std::cin >> m;`
- Da `std::istream` derivano:
  - `ifstream` per leggere da le
  - `istringstream` per leggere da una stringa in memoria. Utile per analizzare una stringa formattata.

`std::istream` è la classe base per tutti gli stream di input. L'operatore get from 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.



Questo diagramma di flusso descrive il funzionamento dell'operazione get from per un intero.

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 una riga per volta, cioè fino al carattere di terminazione linea (`'\n'`). Successivamente si eliminano tutti gli spazi iniziali dal buffer. Se il primo carattere non spazio non è una cifra (per esempio è una lettera) si produce un errore (recuperabile). Altrimenti si leggono e si eliminano dal buffer tutte le cifre, finché non si incontra la fine del buffer o un carattere non-cifra (per esempio uno spazio o una lettera). Dopodiché si restituisce il numero letto e si termina l'algoritmo.

Si noti che l'algoritmo può terminare anche se il buffer non è stato letto ed eliminato tutto. 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`, etc. In particolare tutti scartano gli spazi iniziali e terminano la lettura del buffer quando questo finisce o al primo carattere *non interpretabile* (cioè uno spazio o una lettera nel caso di tipi numerici, etc.).

Funzionamento del get from

```
int i1, i2, i3;

char s1[100];
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 memory 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.

Funzionamento del get from

- L'operatore get from applicato a stringhe acquisisce *parole*.

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

```
␣␣␣␣pluto␣␣␣␣␣␣pippo
pluto
```

- Per acquisire effettivamente *stringhe* bisogna invocare il metodo a basso livello `getline()`.

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

```
␣␣␣␣pluto␣␣␣␣␣␣pippo
␣␣␣␣pluto␣␣␣␣␣␣pippo
```

15

L'operatore get from 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 la funzione `getline()`. `getline()` è uno dei metodi per input *non formattato*. Il metodo `getline()` non ha problemi di overflow, in quanto prende in ingresso la dimensione della stringa.

Esempio – Controllo dell'input

```
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()) {
8           cerr << "An error occurred in stdin" << 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, per esempio l'utente inserisce delle lettere anziché delle cifre, viene stampato un errore a video.

Esempio – Controllo dell'input

```
1   int main(){
2       int age;
3       cout << "How old are you?";
4       cin >> age;
5       while(cin.fail()){
6           if(cin.bad()){
7               cerr << "Stdin error!" << endl;
8               exit(-1);
9           }
10          cout << "That's not a number!" << endl;
11          cin.clear();
12          cin.ignore(100, '\n');
13          cout << "How old are you? ";
14          cin >> age;
15      }
16      cout<<"You are "<<age<<" years old"<<endl;
17  }
```

17

Questo programma è l'estensione di quello precedente, in cui si recupera l'errore logico di aver inserito lettere anziché cifre. Se l'errore è irrecuperabile (`bad()`) dobbiamo terminare il programma. Se invece l'errore è logico stampiamo un avvertimento a video, puliamo la condizione di errore (`clear()`), svuotiamo il buffer (`ignore()`) e ripetiamo la domanda.

Input non formattato

- L'operatore get from, oltre a leggere dallo stream, fa una serie aggiuntiva di operazioni:
  - *Eliminazione* degli spazi
  - *Divisione* tra un campo e il successivo (`120pluto`)
  - *Formattazione*: conversione da cifre a rappresentazione binaria (nel caso degli interi)
- Se non si vuole fare queste operazioni, bisogna ricorrere ai metodi *non formattati*. Se trovano il buffer vuoto acquisiscono una riga.
  - `char ch; cin.get(ch);` → Legge un carattere dal buffer.
  - `char p[100]; cin.getline(p, 100);` → Legge fino al '`\n`' (alpiù 99 caratteri). Mette '`\0`' dopo il carattere finale. Utile per leggere stringhe.
  - `char p[100]; cin.read(p, 100);` → Legge 100 caratteri dal buffer. Se non ce ne sono li acquisisce. Utile per leggere dagli stream binari.
  - `cin.ignore(100, '\n');` → Elimina dal buffer fino al '`\n`' (alpiù 100 caratteri). Utile per svuotare il buffer di lettura.
- `int i = cin.gcount();` → Restituisce il numero di caratteri letti dall'ultima operazione non formattata. Utile per coordinare più operazioni di `ignore()`.

18

Come visto sopra, l'operatore get from opera ad alto livello, ovvero fa delle operazioni aggiuntive rispetto al semplice input. Elimina gli spazi, divide tra un campo ed il successivo (si ricordi l'esempio precedente di "`120pluto`") e formatta i dati, ovvero trasforma da cifre ASCII a rappresentazione a complemento a due.

Se non vogliamo tutte queste funzionalità dobbiamo ricorrere a metodi *non formattati*, che operano a più basso livello. Alcuni sono questi: `get()` legge un carattere (eventualmente uno spazio) dal buffer (se il buffer è vuoto acquisisce una riga). `getline()` legge dal buffer fino al carattere di fine linea o fino al 99esimo carattere. Scrive tutto nel buffer e mette un `'\0'` alla fine. È utile per leggere una riga di input. `read()` legge dal buffer 100 caratteri e li mette nel buffer. Non mette il `'\0'`. È utile per leggere da stream binari (file, etc.). `ignore()` elimina dal buffer fino al `'\n'` o fino al 100esimo carattere. È utile per recuperare errori logici, se si vuole eliminare dei caratteri spuri che l'utente ha inserito. `gcount()` restituisce i caratteri letti dall'ultima operazione non formattata. è utile per coordinare più chiamate di `ignore()`. Supponiamo infatti di dover eliminare dal buffer un'intera riga, indipendentemente dalla sua lunghezza. Se io chiamo `ignore(100,' \n')` e il buffer contiene 120 caratteri non riesco a svuotarlo tutto. Quindi, controllo il valore di `gcount()`. Se è uguale a 100 vuol dire che devo chiamare una nuova `ignore()`.

Scrittura su file
Lettura da file

Esempio – Scrittura su file

```
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  }
```


Questo programma scrive su un file 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 (`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 librearia `<fstream>` (senza .h).

Esempio – Lettura da file

```
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 da un file e lo stampa a video. L'intero dev'essere memorizzato nel file in cifre ASCII. Successivamente chiude il file.

Operazioni sui file

- Nella classe `ios` sono definiti, tramite enumerati varie modalità di apertura di un file:
  - `ios::in`, `ios::out`, `ios::app` per lettura, scrittura, aggiunta
  - `ios::binary` per I/O binario (gli operatori >> e << non effettuano conversioni di tipo)
  - `ios::nocreate` fa fallire l'apertura se il file non esiste (utile per modalità di aggiunta)
  - `ios::noreplace` fa fallire l'apertura se il file esiste (utile per modalità di scrittura)
- `ofstream f("out.txt", ios::out | ios::noreplace)` crea uno stream che scrive sul file "out.txt" in modalità scrittura+no replace.
- Il metodo `close()` provvede a chiudere il file associato allo stream. È invocato automaticamente dal distruttore.
- Le funzioni:
  - `outfile.write(buf, 100);`
  - `infile.read(buf, 100);`

  sono utili per I/O *non formattato* di caratteri (invece di usare gli operatori << e >>).

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.

Per i file binari, sono molto utili i metodi `read()` e `write()`, che acquisiscono input non formattati.

Esempio – Lettura/scrittura da file

```
1   #include <fstream>
2   using namespace std;
3   int main(){
4         ifstream ingr;
5         ofstream usc;
6         ingr.open("source.txt", ios::in);
7         if (ingr.fail()) exit(-1);
8         usc.open("dest.txt", ios::out);
9         if (usc.fail()) exit(-1);
10        char app;
11        ingr.get(app); if (ingr.bad()) exit(-1);
12        while (!ingr.eof()){
13              usc.put(app); if (usc.bad()) exit(-1);
14              ingr.get(app); if (ingr.bad()) exit(-1);
15        }
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 carattere per carattere (`get()`) e si scrive in uscita (`put()`) fino alla fine del file in ingresso. Infine si chiudono entrambi i file.

Esempio – Lettura/scrittura da file (2)

```
1   #include <fstream>
2   using namespace std;
3   int main(){
4         ifstream ingr;
5         ofstream usc;
6         ingr.open("source.txt", ios::in|ios::binary);
7         usc.open("dest.txt", ios::out|ios::binary);
8         if (ingr.fail() || usc.fail()) exit(-1);
9         char app[4096];
10        int letti = ingr.read(app, 4096);
11        if (ingr.bad()) exit(-1);
12        while (!ingr.eof()){
13              usc.write(app, letti); if (usc.bad()) exit(-1);
14              letti = ingr.read(app, 4096);
15              if (ingr.bad()) exit(-1);
16        }
17        usc.write(app, letti); if(usc.bad()) exit(-1);
18        ingr.close();
19        usc.close();
20  }
```


La soluzione precedente è alquanto lenta, in quanto copia il contenuto un byte alla volta. Una soluzione più efficiente è questa. Si alloca un buffer con cui si leggono (`read()`) e si scrivono (`write()`) 4096 byte alla volta. Infine, si scrivono i byte che avanzano e si chiudono i due file.

Gestione del cursore

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

  e quello di output:
  - `long ostream::tellp();`
  - `ostream& ostream::seekp(long 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`.

Bufferizzazione

- L'output è bufferizzato: ciò significa che il buffer viene svuotato e avviene l'effettiva scrittura solamente quando viene inviato nello stream un carattere di fine linea.

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

- Lo svuotamento del buffer può essere imposto tramite la funzione `cout.flush()` o con il *manipolatore* `flush`.

```
1  long tm = time(NULL) + 5;
2  cout << "Wait 5 seconds... " << flush;
3  while (time(NULL) < tm);
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`	Svuota il buffer
`endl`	Aggiunge un carattere di fine linea e svuota il buffer
`hex`	Gli interi vengono formattati in base 16
`oct`	Gli interi vengono formattati in base 8
`dec`	Gli interi vengono formattati in base 10

```
cout << hex << 100 << dec << flush;
```

Output a video:

```
64
```

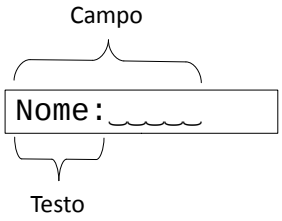

I *manipolatori* sono degli oggetti che possono essere usati come argomenti del get from e del put to e che modificano il loro comportamento. Sopra ne vediamo alcuni esempi. `flush` svuota il buffer. `endl` aggiunge un carattere di fine linea e svuota il buffer. `hex` attiva la codifica esadecimale,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

Manipolatori

- 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 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.

Manipolatori

- Manipolatori di campo

Manipolatore	Significato
`setw(w)`	Setta l'ampiezza del campo (default = 0)
`setfill(c)`	Setta il carattere di riempimento (default = *spazio*)
`left`	Il testo viene allineato a sinistra nel campo
`right`	Il testo viene allineato a destra nel campo (default)

```
cout << setw(10) << left << "Nome:" << flush;
```

Output a video:

```
Nome:␣␣␣␣␣
```


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.

Eccezioni – Esempio

```
1   #include <iostream>
2   using namespace std;
3   int main(){
4         int i;
5         int vett[5]; // ecc: con failbit o eofbit
6         cin.exceptions(ios::failbit | ios::eofbit);
7         try{
8               for (i = 0; i < 5; i++) cin >> vett[i];
9         }
10        catch (ios_base::failure){
11              if (cin.eof())
12                    cerr << "End of cin" << endl ;
13              else
14                    cerr << "Read error" << endl ;
15        }
16  }
```

- `void ios::exceptions(int exc)` fissa i casi in cui sollevare l'eccezione

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 `std::string` non è implementata tramite un 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.
- Ai nostri fini, una `std::string` e una sequenza di `char`. Essendo un tipo di dati concreto, e possibile istanziare, copiare ed assegnare un oggetto di tipo `std::string` come un qualsiasi oggetto primitivo.
- Un oggetto di tipo `std::string` viene allocato, riallocato e deallocato 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.

Esempio di utilizzo delle `std::string`

```
1   #include <string>
2   #include <iostream>
3   int main(){
4         std::string s1 = "Hello, ";
5         std::string s2;
6         s2 = " world! ";
7         std::string s3;
8         std::cin >> s3;
9         std::string s4 = s1 + s2 + s3 + "\n";
10        std::cout << s4;
11  }
```

Output a video:

```
pippo
Hello, world! pippo
```

- Le `std::string` interagiscono in maniera naturale con gli stream di ingresso/uscita della STL

Questo programma mostra alcune operazioni possibili con le stringhe. L'operatore + indica la concatenazione.

Differenze tra `std::string` e stringhe C

- Allocazione, assegnamento, concatenazione, confronto:

```c
#include <string.h>
int main()
{
	char s1[] = "abc";
	char s2[7];
	strcpy(s2, s1);
	strcat(s2, s1);
	if(strcmp(s2, s1) == 0)
		// ...
}
```

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

- Input da stream:

```c
char s1[100];
cin >> s1;
```

```c
string s1;
cin >> s1;
```


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 (=) assegna il *valore* di una stringa (cioè il suo contenuto) ad un'altra. 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.

Metodi principali

- Creazione di un oggetto `std::string`
  - `string::string()`
  - `string::string(const string& s)`
  - `string::string(const char* s)`
- Metodi principali:
  - `const char*  string::c_str() const`
  - `char&  string::operator[](size_type pos)`
  - `unsigned int  string::length() const`
  - `unsigned int  string::capacity() const`
  - `void  string::reserve(unsigned int)`

```
string s1;
cin >> s1;
const char* p = s1.c_str();
s1 += "pippo";
cout << p;
```

Errore!
Il puntatore p
non è più valido!

Il costruttore di default crea un stringa vuota. Il costruttore di copia fa una copia per valore della stringa. Il costruttore di conversione converte da stringa C a stringa C++.

Il metodo `c_str()` restituisce un puntatore ad una stringa C *costante* che ha lo stesso valore della stringa C++. Viene usato come una sorta di conversione inversa da stringa C++ a stringa C. Con una differenza: se si memorizza il puntatore `c_str()` e poi si modifica la `string`, il puntatore viene invalidato.

L'operatore parentesi quadra restituisce un riferimento al carattere di indice `pos`. Tale carattere può essere letto o modificato, analogamente a quanto succede con le stringhe C. Gli indici partono da 0 (*zero-based index*). `length()` restituisce la lunghezza della stringa. `capacity()` restituisce la grandezza della memoria interna allocata per la stringa (sempre maggiore o uguale alla lunghezza della stringa). Il metodo `reserve()` serve a 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.

Manipolazione di stringhe

- Sono definiti l'operatore di concatenazione (+) e di concatenazione e assegnamento (+=), tra due `string` o tra una `string` e una stringa C
  - `s3 = s1 + s2;`
  - `char p[]="ciao"; s3 = s1 + p;`
  - `s2 += s1;`
  - `char p[]="ciao"; s2 += p;`
- 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 (per valore) di una sottostringa.
  - `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 e ricerche

- 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; 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`.
  - `i = s1.rfind(s2, pos);` → Stessa cosa ma cerca l'*ultima* occorrenza nelle posizioni *minori* di `pos`. Il valore di default di `pos` è `string::npos`.
  - `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`.
  - `i = s1.find_last_of("cde", pos);` → Stessa cosa 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.

Confronti e ricerche

```
1   #include <string>
2   #include <iostream>
3   using namespace std;
4   int main(){
5         string s1;
6         cin >> s1;
7         int pos = 0;
8         do{
9               pos = s1.find("cd", pos);
10              if (pos != string::npos){
11                    cout << "pos = " << pos << endl;
12                    pos++;
13              }
14        }
15        while (pos != string::npos);
16  }
```

```
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 ogni volta dalla posizione dell'ultima occorrenza trovata più uno.

Stringhe e I/O

- Sono definito gli operatori >> e << per input e output di stringhe su stream:
  - `std::ostream& operator<<(std::ostream&, const std::string&);`
  - `std::istream& operator>>(std::istream&, std::string&);`
- L'operatore >> fa l'input di singole *parole*!
  - `getline(cin, s1);` ← Per acquisire una stringa fino al '\n' si usa `getline()` (metodo *globale*).

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

Scambio di due stringhe

- Versione inefficiente (fa tre copie di stringa):

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

- Versione efficiente (scambia i puntatori interni):

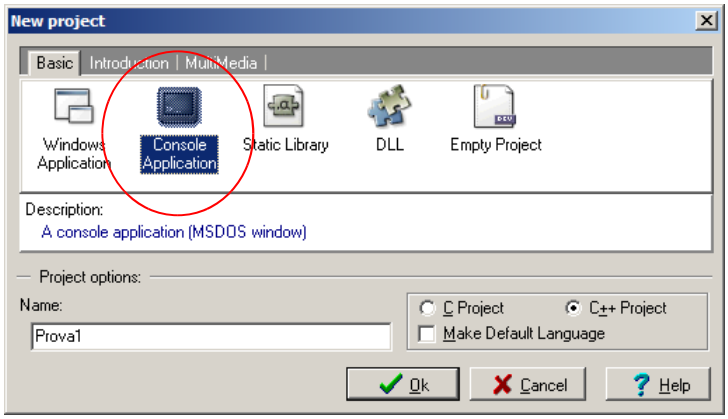
```
swap(s1, s2);
```


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.

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 con bubblesort;
  - 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/