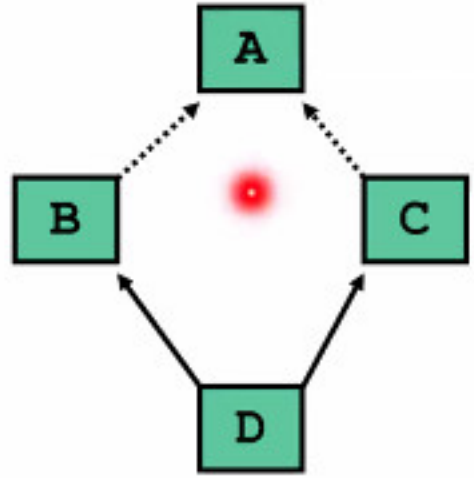
```
class Z {
public:
  Z() {cout << "Z() ";}
  Z(const Z& x) {cout << "Zc ";}
};

class A {
private:
  Z w;
public:
  A() {cout << "A() ";}
  A(const A& x) {cout << "Ac ";}
};

class B: virtual public A {
private:
  Z z;
public:
  B() {cout << "B() ";}
  B(const B& x) {cout << "Bc ";}
};

class C: virtual public A {
private:
  Z z;
public:
  C() {cout << "C() ";}
};

class D: public B, public C {
public:
  D() {cout << "D() ";}
  D(const D& x): C(x) {cout << "Dc ";}
};
```

Cosa stampa?

```
D d1;
D d2 = d1;
```



```
class Z {
public:
  Z() {cout << "Z() ";}
  Z(const Z& x) {cout << "Zc ";}
};

class A {
private:
  Z W;
public:
  A() {cout << "A() ";}
  A(const A& x) {cout << "Ac ";}
};

class B: virtual public A {
private:
  Z z;
public:
  B() {cout << "B() ";}
  B(const B& x) {cout << "Bc ";}
};

class C: virtual public A {
private:
  Z z;
public:
  C() {cout << "C() ";}
};

class D: public B, public C {
public:
  D() {cout << "D() ";}
  D(const D& x): C(x) {cout << "Dc ";}
};
```

Cosa stampa?

D d1;

Z( ) A( ) Z( ) B( ) Z( ) C( ) D( )

```
class Z {
public:
  Z() {cout << "Z() ";}
  Z(const Z& x) {cout << "Zc ";}
};

class A {
private:
  Z W;
public:
  A() {cout << "A() ";}
  A(const A& x) {cout << "Ac ";}
};

class B: virtual public A {
private:
  Z z;
public:
  B() {cout << "B() ";}
  B(const B& x) {cout << "Bc ";}
};

class C: virtual public A {
private:
  Z z;
public:
  C() {cout << "C() ";}
};

class D: public B, public C {
public:
  D() {cout << "D() ";}
  D(const D& x): C(x) {cout << "Dc ";}
};
```

Cosa stampa?

```
D d2=d1;
```

Z() A() Z() B() Zc Dc

Si assuma che A, B, C, D siano quattro classi polimorfe. Si consideri il seguente main().

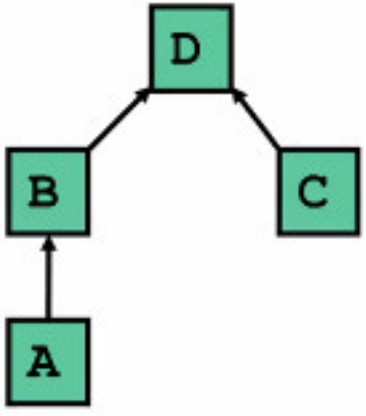
```
main() {
 A a; B b; C c; D d;
 cout << (dynamic_cast<D*>(&c) ? "0 " : "1 ");
 cout << (dynamic_cast<B*>(&c) ? "2 " : "3 ");
 cout << (!(dynamic_cast<C*>(&b)) ? "4 " : "5 ");
 cout << (dynamic_cast<B*>(&a) || dynamic_cast<C*>(&a)  ? "6 " : "7 ");
 cout << (dynamic_cast<D*>(&b) ? "8 " : "9 ");
}
```

Si supponga che tale main() compili ed esegua correttamente. Disegnare i diagrammi di **tutte** le possibili gerarchie per le classi A, B, C, D tali che l'esecuzione del main() provochi la stampa: 0 3 4 6 8.

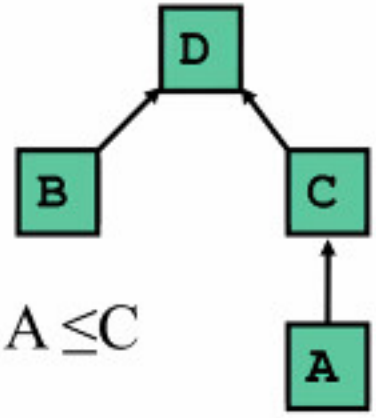
Si assuma che `A`, `B`, `C`, `D` siano quattro classi polimorfe. Si consideri il seguente `main()`.

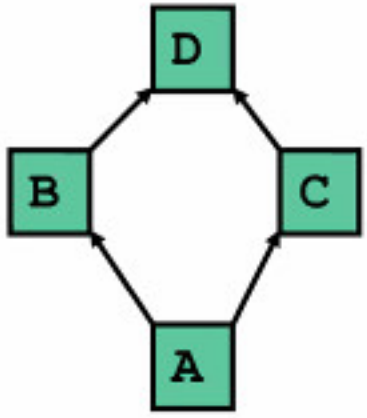
```
main() {
 A a; B b; C c; D d;
 cout << (dynamic_cast<D*>(&c) ? "0 " : "1 ");
 cout << (dynamic_cast<B*>(&c) ? "2 " : "3 ");
 cout << (!(dynamic_cast<C*>(&b)) ? "4 " : "5 ");
 cout << (dynamic_cast<B*>(&a) || dynamic_cast<C*>(&a)  ? "6 " : "7 ");
 cout << (dynamic_cast<D*>(&b) ? "8 " : "9 ");
}
```

Si supponga che tale `main()` compili ed esegua correttamente. Disegnare i diagrammi di **tutte** le possibili gerarchie per le classi `A`, `B`, `C`, `D` tali che l'esecuzione del `main()` provochi la stampa: `0 3 4 6 8`.



$C \leq D$
$C \not\leq B$
$B \not\leq C$
$A \leq B$ or $A \leq C$
$B \leq D$





cppreference.com

Log in / create account

Page | Discussion | View | View source | History

C++ reference

FAQ

Language
- Preprocessor
- Keywords
- Operator precedence
- Escape sequences
- ASCII chart
- Fundamental types

Headers

Concepts

Utilities library
- Type support (basic types, RTTI, type traits)
- Dynamic memory management
- Error handling
- Program utilities
- Date and time
- bitset
- Function objects
- pair
- tuple (C++11)

Strings library
- basic_string
- Null-terminated byte strings
- Null-terminated multibyte strings
- Null-terminated wide strings

Containers library
- array (C++11)
- vector
- deque
- list
- forward_list (C++11)
- set
- multiset
- map
- multimap
- unordered_set (C++11)
- unordered_multiset (C++11)
- unordered_map (C++11)
- unordered_multimap (C++11)
- stack
- queue
- priority_queue

Algorithms library

Iterators library

Numerics library
- Common mathematical functions

Input/output library
- basic_streambuf
- basic_filebuf
- basic_stringbuf
- ios_base
- basic_ios
- basic_istream
- basic_ostream
- basic_iostream
- basic_ifstream
- basic_ofstream
- basic_fstream
- basic_istringstream
- basic_ostringstream
- basic_stringstream
- I/O manipulators
- C-style I/O

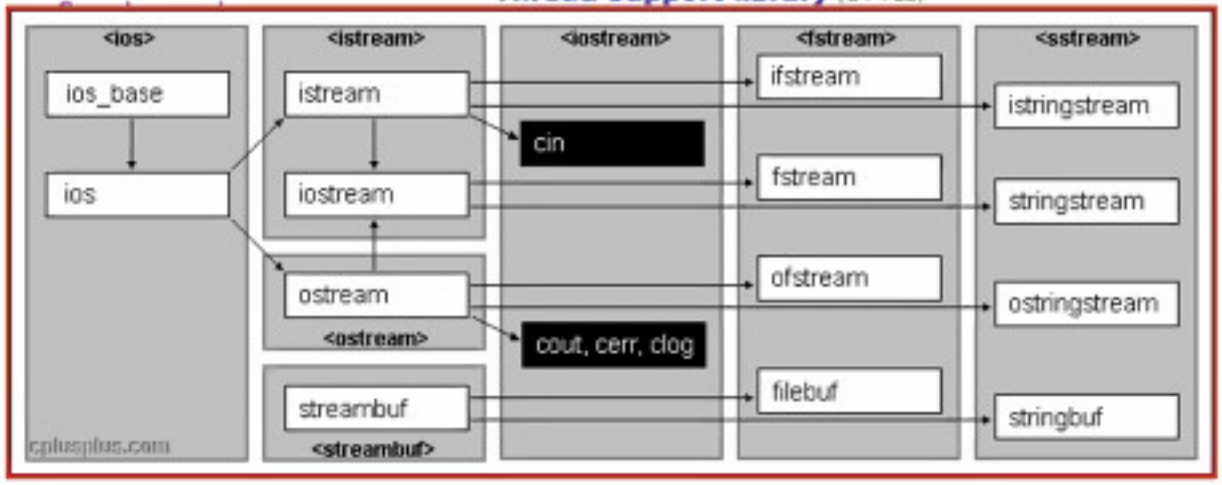
documentazione (molto buona)

Localizations library

Regular expression library (C++11)

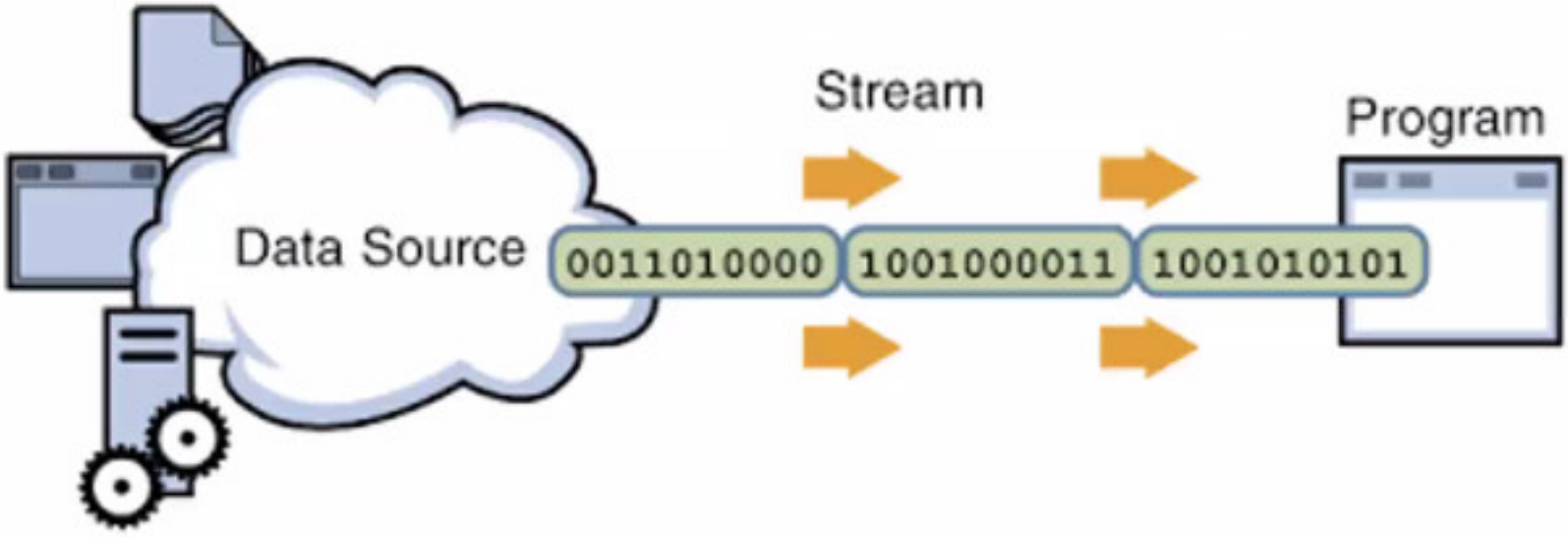
Atomic operations library (C++11)

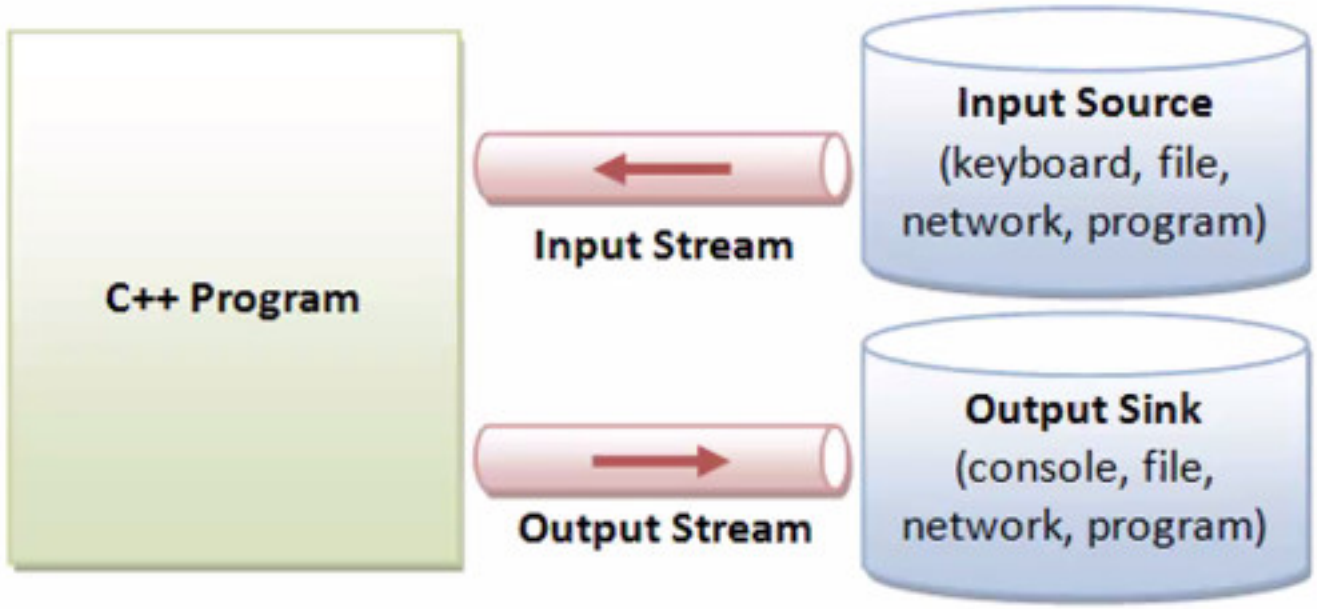
Thread support library (C++11)



I/O mediante l'astrazione di dispositivo di I/O detta **stream**

Stream = "sequenza (non limitata) di celle ciascuna contenente un byte"





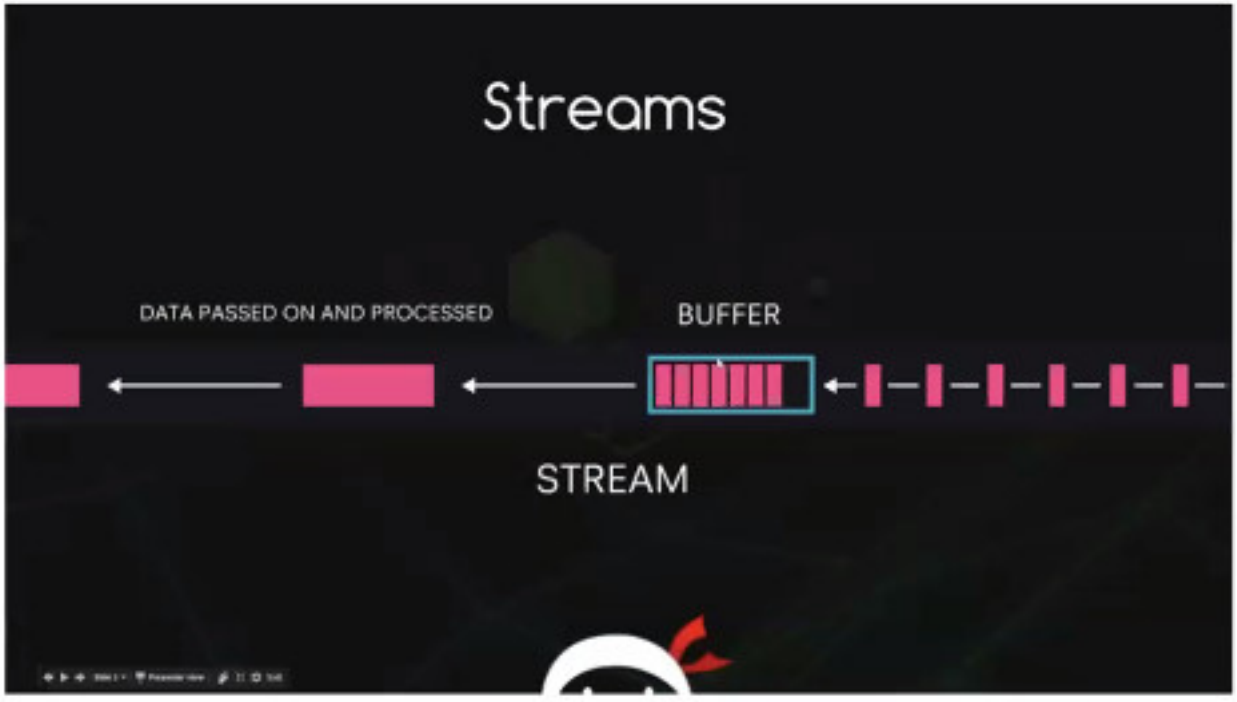
Internal Data Formats:

- Text: char
- int, float, double, etc.

External Data Formats:

- Text in various encodings (US-ASCII, ISO-8859-1, UCS-2, UTF-8, UTF-16, UTF-16BE, UTF16-LE, etc.)
- Binary (raw bytes)

- La posizione delle celle di uno stream **parte da 0**
- I/O effettivo avviene tramite un **buffer** associato allo stream
- Uno stream può trovarsi nello stato di **end-of-file**



Uno stream può trovarsi in 8 ($=2^3$) stati di funzionamento diversi. Lo stato è un intero in [0,7] rappresentato dal campo dati **`state`** della classe base **`ios`** che corrisponde al numero binario

bad fail eof

dove bad, fail ed eof sono dei bit (0 o 1) di stato:

eof==1 ⇔ lo stream è nella **posizione di end-of-file**.

fail==1 ⇔ la precedente operazione sullo stream è fallita: si tratta di un **errore senza perdita di dati**, normalmente è possibile continuare. Ad esempio, ci si aspettava in input un **`int`** e si trova invece un **`double`**.

bad==1 ⇔ la precedente operazione sullo stream è fallita con perdita dei dati: è un **errore fatale/fisico**, normalmente non è possibile continuare. Ad esempio, cerco di accedere ad un file o ad una network connection inesistenti

La classe ios

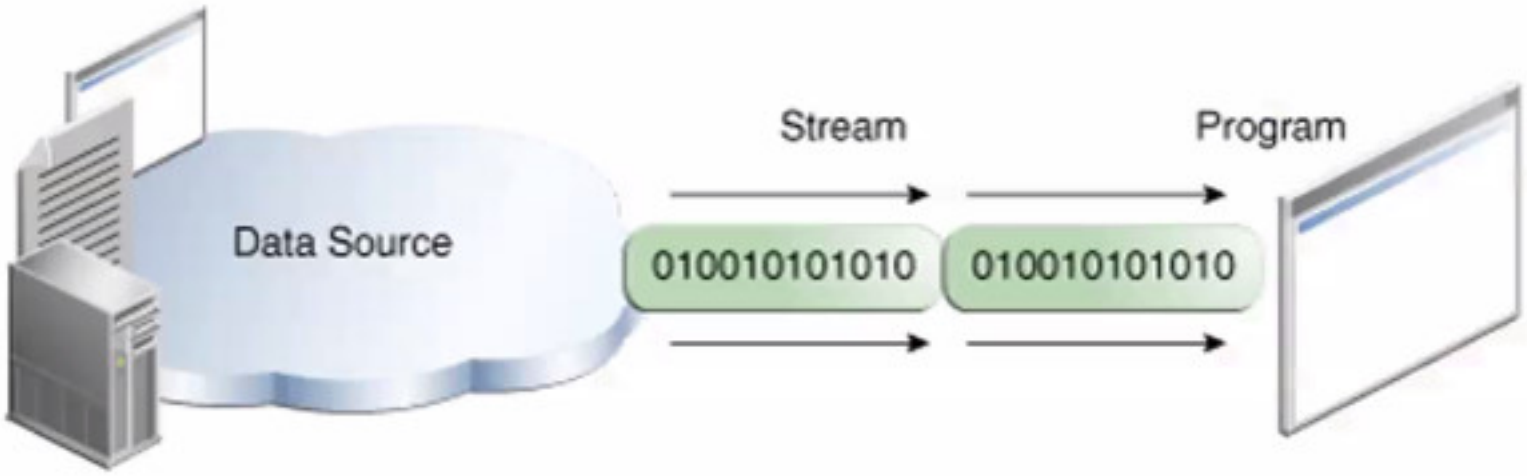
ios (derivata da **ios_base**) è la **classe base astratta "virtuale"** della gerarchia che permette di controllare lo stato di funzionamento di uno stream. Per quanto concerne lo stato di uno stream, la dichiarazione della classe **ios** è la seguente:

```
class ios: public ios_base {
  int state;   // stato dello stream
public:        //       000        001         010         100
  enum io_state {goodbit=0, eofbit=1, failbit=2, badbit=4};
  int good() const;     // ritorna 1 se lo stato è good
  int eof() const;      // ritorna il bit eof
  int fail() const;     // ritorna il bit fail
  int bad() const;      // ritorna il bit bad
  int rdstate() const; // ritorna lo stato come int in [0,7]
  void clear(int i=0); // setta lo stato i
...
};
```

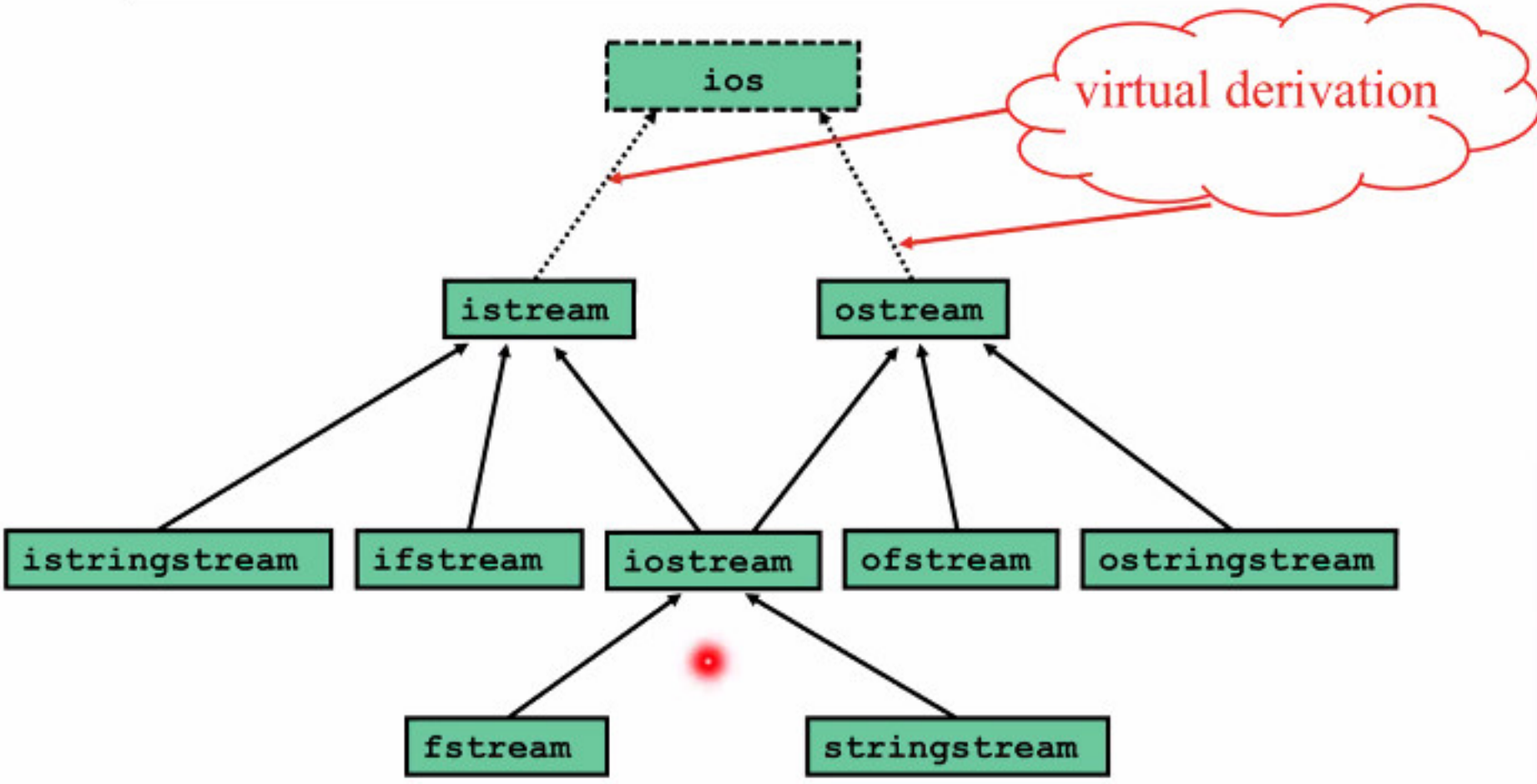
badbit failbit eofbit

La classe istream

Gli oggetti della sottoclasse **`istream`** rappresentano **stream di input**.
`cin` è un oggetto di **`istream`** che rappresenta lo standard input.



Gerarchia di classi per l'I/O



`istream` include l'overloading dell'operatore di input `operator>>` per i tipi primitivi e per gli array di caratteri.

```
class istream: public virtual ios {
public:
  // metodi interni (con istream di invocazione)
  istream& operator>>(bool&);
  istream& operator>>(int&);
  istream& operator>>(double&);
  ...
};
// funzioni esterne in std::
istream& std::operator>>(istream&, char&); // byte
istream& std::operator>>(istream&, char*); // stringhe
```

Tutti gli operatori di input **ignorano le spaziature** (cioè spazi, tab, enter) presenti prima del valore da prelevare.

Quando una **operazione di input fallisce (fail==1) non viene effettuato alcun prelievo** dallo stream e la variabile argomento di `operator>>` non subisce modifiche.

EXAMPLE

`operator>>(double& val)` preleva dallo istream di invocazione una sequenza di caratteri che rispetta la sintassi dei litterali di `double` e converte tale sequenza nella rappresentazione numerica di `double` assegnandolo a `val`. Se la sequenza di caratteri non soddisfa la sintassi prevista per `double`, l'operazione è nulla e l'istream va in uno stato di errore recuperabile: fail=1 e bad=0.

EXAMPLE

`operator>>(double& val)` preleva dallo istream di invocazione una sequenza di caratteri che rispetta la **sintassi dei litterali di `double`** e converte tale sequenza nella rappresentazione numerica di **`double`** assegnandolo a **`val`**. Se la sequenza di caratteri non soddisfa la sintassi prevista per **`double`**, l'operazione è nulla e l'istream va in uno stato di errore recuperabile: fail=1 e bad

floating point literal

Floating point literal defines a compile-time constant whose value is specified in the source file.

Syntax

significand exponent(optional) *suffix*(optional)

Where the *significand* has one of the following forms

digit-sequence	(1)	
digit-sequence .	(2)	
digit-sequence(optional) . *digit-sequence*	(3)	
0x | **0X** *hex-digit-sequence*	(4)	(since C++17)
0x | **0X** *hex-digit-sequence* .	(5)	(since C++17)
0x | **0X** *hex-digit-sequence*(optional) . *hex-digit-sequence*	(6)	(since C++17)

1) *digit-sequence* representing a whole number without a decimal separator, in this case the exponent is not optional: `1e10`, `1e-5L`
2) *digit-sequence* representing a whole number with a decimal separator, in this case the exponent is optional: `1.`, `1.e-2`
3) *digit-sequence* representing a fractional number. The exponent is optional: `3.14`, `.1f`, `0.1e-1L`
4) Hexadecimal *digit-sequence* representing a whole number without a radix separator. The exponent is never optional for hexadecimal floating-point literals: `0x1ffp10`, `0X0p-1`
5) Hexadecimal *digit-sequence* representing a whole number with a radix separator. The exponent is never optional for hexadecimal floating-point literals: `0x1.p0`, `0xf.p-1`
6) Hexadecimal *digit-sequence* representing a fractional number with a radix separator. The exponent is never optional for hexadecimal floating-point literals: `0x0.123p-1`, `0xa.bp10l`

The *exponent* has the form

e | **E** *exponent-sign*(optional) *digit-sequence*	(1)	
p | **P** *exponent-sign*(optional) *digit-sequence*	(2)	(since C++17)

1) The exponent syntax for a decimal floating-point literal
2) The exponent syntax for hexadecimal floating-point literal

exponent-sign, if present, is either **+** or **-**

suffix, if present, is one of **f**, **F**, **l**, or **L**. The suffix determines the type of the floating-point literal:

- (no suffix) defines `double`
- **f** **F** defines `float`
- **l** **L** defines `long double`

Optional single quotes(') can be inserted between the digits as a separator, they are ignored when compiling. (since C++14)

EXAMPLE

Grammatica che definisce la sintassi dei litterali in virgola mobile

floating point literal

Floating point literal defines a compile-time constant whose value is specified in the source file.

Syntax

significand *exponent*(optional) *suffix*(optional)

Where the *significand* has one of the following forms

digit-sequence	(1)	
digit-sequence **.**	(2)	
digit-sequence(optional) **.** *digit-sequence*	(3)	
0x | **0X** *hex-digit-sequence*	(4)	(since C++17)
0x | **0X** *hex-digit-sequence* **.**	(5)	(since C++17)
0x | **0X** *hex-digit-sequence*(optional) **.** *hex-digit-sequence*	(6)	(since C++17)

1) *digit-sequence* representing a whole number without a decimal separator, in this case the exponent is not optional: `1e10`, `1e-5L`
2) *digit-sequence* representing a whole number with a decimal separator, in this case the exponent is optional: `1.`, `1.e-2`
3) *digit-sequence* representing a fractional number. The exponent is optional: `3.14`, `.1f`, `0.1e-1L`
4) Hexadecimal *digit-sequence* representing a whole number without a radix separator. The exponent is never optional for hexadecimal floating-point literals: `0x1ffp10`, `0X0p-1`
5) Hexadecimal *digit-sequence* representing a whole number with a radix separator. The exponent is never optional for hexadecimal floating-point literals: `0x1.p0`, `0xf.p-1`
6) Hexadecimal *digit-sequence* representing a fractional number with a radix separator. The exponent is never optional for hexadecimal floating-point literals: `0x0.123p-1`, `0xa.bp10l`

The *exponent* has the form

e | **E** *exponent-sign*(optional) *digit-sequence*	(1)	
p | **P** *exponent-sign*(optional) *digit-sequence*	(2)	(since C++17)

1) The exponent syntax for a decimal floating-point literal
2) The exponent syntax for hexadecimal floating-point literal

exponent-sign, if present, is either **+** or **-**

suffix, if present, is one of **f**, **F**, **l**, or **L**. The suffix determines the type of the floating-point literal:

- (no suffix) defines `double`
- **f** **F** defines `float`
- **l** **L** defines `long double`

Optional single quotes(**'**) can be inserted between the digits as a separator, they are ignored when compiling. (since C++14)

`operator>>(double& val)` preleva dallo istream di invocazione una sequenza di caratteri che rispetta la sintassi dei litterali di **`double`** e converte tale sequenza nella rappresentazione numerica di **`double`** assegnandolo a **`val`**. Se la sequenza di caratteri non soddisfa la sintassi prevista per **`double`**, l'operazione è nulla e l'istream va in uno stato di errore recuperabile: fail==1 e bad==0.

Extract formatted input

This operator (>>) applied to an input stream is known as *extraction operator*. It is overloaded as a member function for:

(1) arithmetic types

Extracts and parses characters sequentially from the stream to interpret them as the representation of a value of the proper type, which is stored as the value of *val*.

Definire un overloading di **`operator>>`** per qualche classe C significa dare un significato alla conversione

sequenza di byte ⇒ oggetto di C

cioè significa fare del **parsing** di una sequenza di byte di input secondo le regole di rappresentazione sintattica degli oggetti di C

Parsing

Da Wikipedia, l'enciclopedia libera.

In informatica, il **parsing**, **analisi sintattica** o **parsificazione** è un processo che analizza un flusso continuo di dati in ingresso (input, letti per esempio da un file o una tastiera) in modo da determinare la sua struttura grazie ad una data grammatica formale. Un **parser** è un programma che esegue questo compito.

Di solito i parser non sono scritti a mano, ma realizzati attraverso dei generatori di parser.

EXAMPLE

`operator>>(double& val)` preleva dallo istream di invocazione una sequenza di caratteri che rispetta la sintassi dei litterali di **`double`** e converte tale sequenza nella rappresentazione numerica di **`double`** assegnandolo a **`val`**. Se la sequenza di caratteri non soddisfa la sintassi prevista per **`double`**, l'operazione è nulla e l'istream va in uno stato di errore recuperabile: fail==1 e bad==0.

Extract formatted input

This operator (>>) applied to an input stream is known as *extraction operator*. It is overloaded as a member function for:

(1) arithmetic types

Extracts and parses characters sequentially from the stream to interpret them as the representation of a value of the proper type, which is stored as the value of *val*.

flag	error
`eofbit`	The input sequence has no more characters available (*end-of-file* reached).
`failbit`	Either no characters were extracted, or the characters extracted could not be interpreted as a valid value of the appropriate type.
`badbit`	Error on stream (such as when this function catches an exception thrown by an internal operation). When set, the integrity of the stream may have been affected.

EXAMPLE

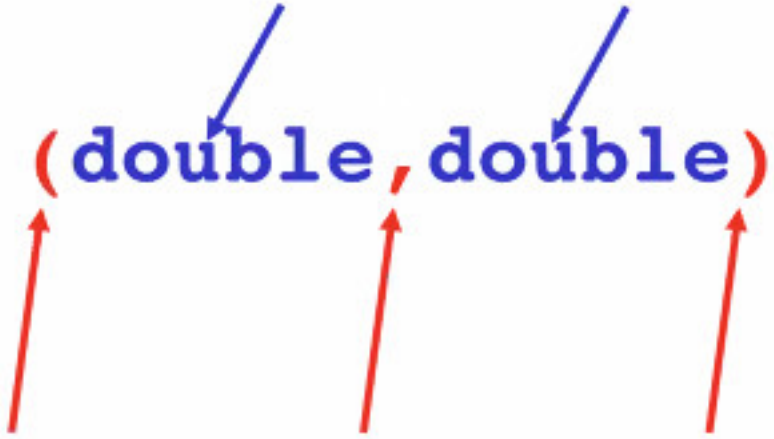
`operator>>(double& val)` preleva dallo istream di invocazione una sequenza di caratteri che rispetta la sintassi dei litterali di `double` e converte tale sequenza nella rappresentazione numerica di `double` assegnandolo a `val`. Se la sequenza di caratteri non soddisfa la sintassi prevista per `double`, l'operazione è nulla e l'istream va in uno stato di errore recuperabile: fail==1 e bad==0.

`operator>>(istream& is,char* s)` preleva dallo istream `is` una sequenza di caratteri fino ad incontrare il carattere spazio (che non viene prelevato), a questa sequenza viene aggiunto in coda il carattere nullo (codice ASCII 0) e viene quindi fatta puntare dal puntatore `s`.

Quando una operazione di input fallisce (fail=1) non viene effettuato alcun prelievo dallo stream e la variabile argomento di `operator>>` non subisce modifiche.

EXAMPLE

Parsing di un oggetto punto nel piano reale rappresentato testualmente in forma cartesiana come



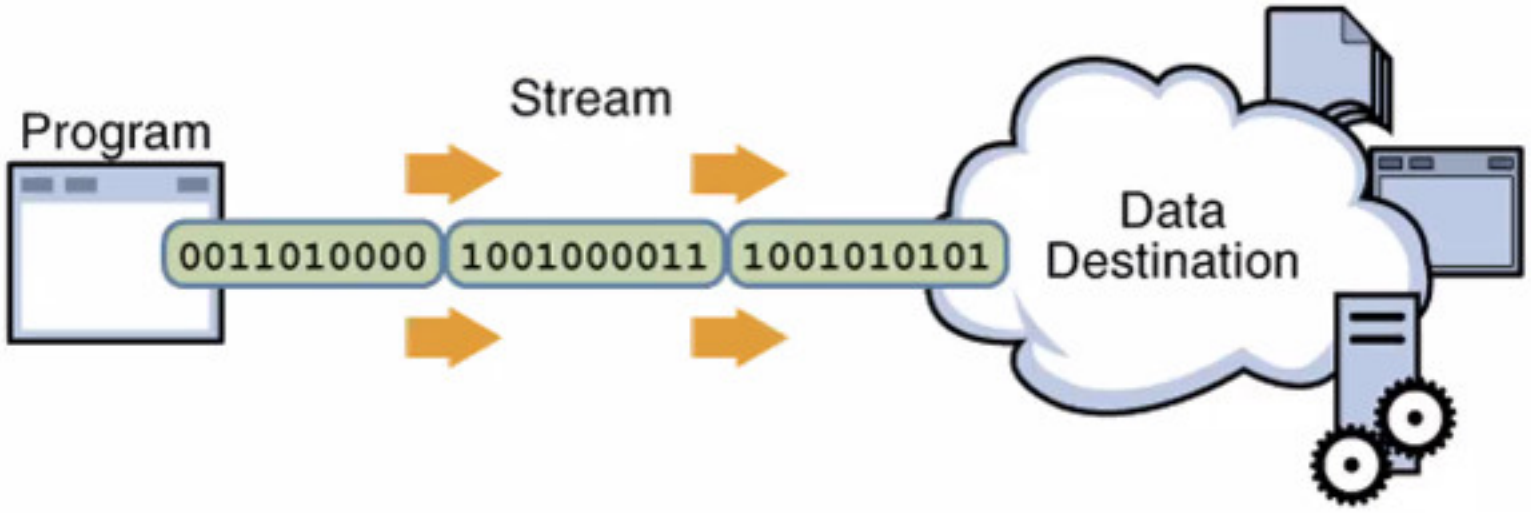
```
class Punto {
  friend istream& operator>>(istream&, Punto&);
  // legge nel formato (x1,x2): rappresentazione testuale di Punto
private:
  double x, y;
};

// ALGORITMO DI PARSING
istream& operator>>(istream& in, Punto& p) {
  char cc; in >> cc; // std::operator>>(istream&,char&)
  if (cc=='q') return in; // carattere q per uscire
  if (cc != '(') { in.clear(ios::failbit); return in;}
  else {









}
```

```
class Punto {
  friend istream& operator>>(istream&, Punto&);
  // legge nel formato (x1,x2): rappresentazione testuale di Punto
private:
  double x, y;
};

// ALGORITMO DI PARSING
istream& operator>>(istream& in, Punto& p) {
  char cc; in >> cc; // std::operator>>(istream&,char&)
  if (cc=='q') return in; // carattere q per uscire
  if (cc != '(') { in.clear(ios::failbit); return in;}
  else {
    in >> p.x;          // istream::operator>>(double&)
    if(!in.good()) { in.clear(ios::failbit); return in;}
    in >> cc;
    if (cc != ',') { in.clear(ios::failbit); return in;}
    else {
      in >> p.y;        // istream::operator>>(double&)
      if(!in.good()) { in.clear(ios::failbit); return in;}
      in >> cc;
      if (cc != ')') { in.clear(ios::failbit); return in;}
    }
  }
  return in;
}
```

```
#include "Punto.h"
using std::cin; using std::cout;

int main() {
 Punto p;
 cout <<"Inserisci un punto nel formato (x,y) ['q' per uscire]\n";
 while(cin.good()) { // while(stato == 0)
  cin >> p;
  if(cin.fail()) {
    cout << "Input non valido, ripetere!\n";
    cin.clear(ios::goodbit);
    char c=0;
    // 10 è il codice ASCII del carattere newline
    while(c!=10) { cin.get(c); } // svuota cin, get() per input binario
    cin.clear(ios::goodbit);
  }
  else cin.clear(ios::eofbit); // stato 1
 }
}
```

La classe ostream

Gli oggetti della sottoclasse ostream rappresentano **stream di output.**
`cout` e `cerr` sono oggetti di ostream (standard output ed error).



EXAMPLE

`operator>>(double& val)` preleva dallo istream di invocazione una sequenza di caratteri che rispetta la sintassi dei litterali di `double` e converte tale sequenza nella rappresentazione numerica di `double` assegnandolo a `val`. Se la sequenza di caratteri non soddisfa la sintassi prevista per `double`, l'operazione è nulla e l'istream va in uno stato di errore recuperabile: fail==1 e bad==0.

`operator>>(istream& is,char* s)` preleva dallo istream `is` una sequenza di caratteri fino ad incontrare il carattere spazio (che non viene prelevato), a questa sequenza viene aggiunto in coda il carattere nullo (codice ASCII 0) e viene quindi fatta puntare dal puntatore `s`.

Quando una operazione di input fallisce (fail=1) non viene effettuato alcun prelievo dallo stream e la variabile argomento di `operator>>` non subisce modifiche.

Definire un overloading di **`operator>>`** per qualche classe C significa dare un significato alla conversione

sequenza di byte ⇒ oggetto di C

cioè significa fare del **parsing** di una sequenza di byte di input secondo le regole di rappresentazione sintattica degli oggetti di C

Parsing

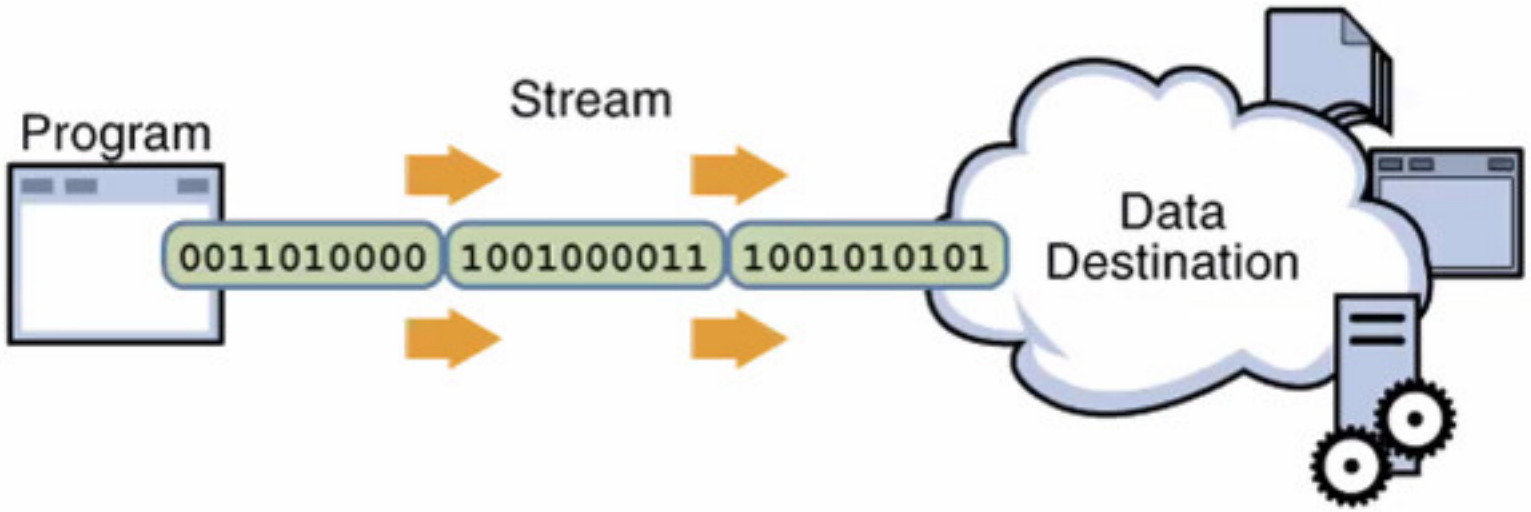
Da Wikipedia, l'enciclopedia libera.

In informatica, il **parsing**, **analisi sintattica** o **parsificazione** è un processo che analizza un flusso continuo di dati in ingresso (input, letti per esempio da un file o una tastiera) in modo da determinare la sua struttura grazie ad una data grammatica formale. Un **parser** è un programma che esegue questo compito.

Di solito i parser non sono scritti a mano, ma realizzati attraverso dei generatori di parser.

La classe ostream

Gli oggetti della sottoclasse ostream rappresentano **stream di output.**
`cout` e `cerr` sono oggetti di ostream (standard output ed error).



La classe ostream include l'overloading dell'operatore di output **`operator<<`** per i tipi primitivi e per gli array di caratteri costanti.

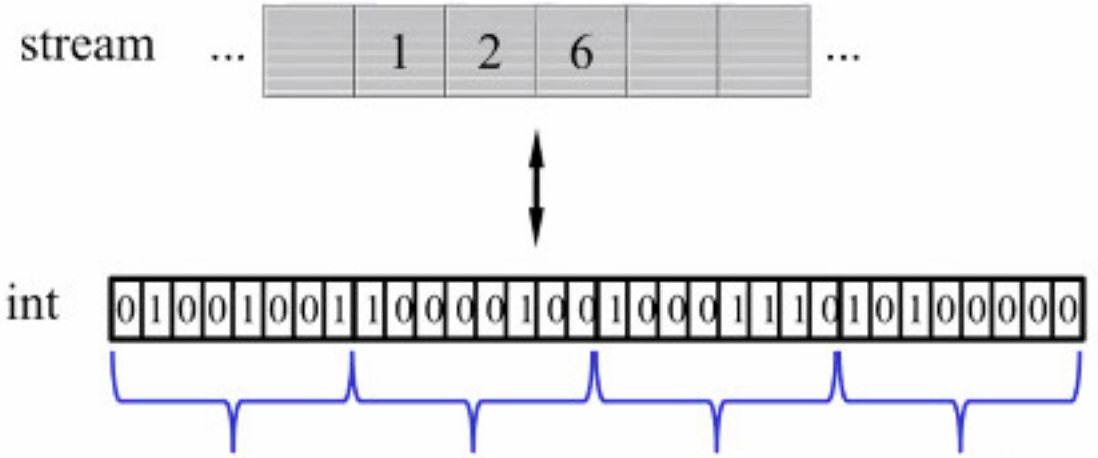
```
class ostream: public virtual ios {
public:
  // metodi interni (con ostream di invocazione)
  ostream& operator<<(bool);
  ostream& operator<<(int);
  ostream& operator<<(double);
  ...
};
// funzioni esterne
ostream& std::operator<<(ostream&,char);          // byte
ostream& std::operator<<(ostream&,const char*); // stringhe
```

Questi operatori convertono valori di tipo primitivo in sequenze di caratteri che vengono scritti (immessi) nelle celle dell'ostream di invocazione. Per quanto riguarda l'output di stringhe, i caratteri della stringa vengono scritti nell'ostream fino al carattere nullo escluso.

I/O testuale e binario

L'input/output sugli stream tramite gli operatori di input/output >> e << considerano gli stream nel cosiddetto ***formato testo***.

```
T x;
cin >> x;
/* conversione dalla rappresentazione testuale di T
   alla rappresentazione binaria in memoria di T */
cout << x;
/* conversione dalla rappresentazione binaria in
memoria di T alla rappresentazione testuale di T */
```



I/O testuale e binario

Quindi, l'informazione da leggere o scrivere su uno stream deve avere una **natura testuale**. Spesso ciò non è vero (almeno in modo naturale). In questo caso, possiamo considerare lo stream in ***formato binario***, cioè tutti i singoli caratteri dello stream vengono trattati allo stesso modo **senza alcuna interpretazione**.

L'**input binario** da uno istream, cioè carattere per carattere (byte per byte senza interpretazione per i byte) può essere fatto tramite alcuni metodi di "get()" di istream.

```
class istream: public virtual ios {
public:
  int get();
  istream& get(char& c);
  istream& read(char* p, int n);
  istream& ignore(int n=1, int e = EOF);
}
```

Il metodo **`int get()`** preleva un singolo carattere (1 byte) dall'istream di invocazione e lo restituisce convertito ad intero in [0,255]. Se si è tentato di leggere EOF ritorna -1.

Il metodo **`get(char& c)`** invece memorizza in **`c`** il carattere prelevato, se questo esiste.

Il metodo **`read(char* p, int n)`** preleva dall'istream di invocazione **`n`** caratteri, a meno che non incontri prima EOF, e li memorizza in una stringa puntata da **`p`**.

Il metodo **`ignore(int n,int e=EOF)`** effettua il prelievo di **`n`** caratteri ma non li memorizza.

L'**output binario** su uno ostream può essere fatto tramite i seguenti metodi di "put()" di ostream.

```
class ostream: public virtual ios {
public:
  ostream& put(char c);
  ostream& write(const char* p, int n);
...
}
```

Il metodo `put(char c)` scrive il carattere `c` nello ostream di invocazione.

Il metodo `write(const char* p, int n)` scrive sullo ostream di invocazione i primi `n` caratteri della stringa puntata da `p`.

Stream di file

Gli stream associati a file sono oggetti delle classi **`ifstream`**, **`ofstream`** e **`fstream`**. Sono disponibili diversi costruttori (vedere documentazione), i più comuni dei quali sono:

```
ifstream(const char* nomefile, int modalita=ios::in);
ofstream(const char* nomefile, int modalita=ios::out);
fstream(const char* nomefile, int modalita=ios::in | ios::out);
```

La stringa **`nomefile`** è il nome del file associato allo stream, mentre le modalità di apertura dello stream sono specificate da un tipo enum nella classe base **`ios`**

```
class ios {
public:
  enum openmode {
  in,              // apertura in lettura
  out,             // apertura in scrittura
  ate,             // spostamento a EOF dopo l'apertura
  app,             // spostamento a EOF prima di ogni write
  trunc,           // erase file all'apertura
  binary,          // apertura in binary mode, default text mode
  };
...
}
```

member constant	opening mode
app	(**app**end) Set the stream's position indicator to the end of the stream before each output operation.
ate	(**at** e**n**d) Set the stream's position indicator to the end of the stream on opening.
binary	(**binary**) Consider stream as binary rather than text.
in	(**in**put) Allow input operations on the stream.
out	(**out**put) Allow output operations on the stream.
trunc	(**trunc**ate) Any current content is discarded, assuming a length of zero on opening.

Le modalità di apertura di uno stream su file possono essere combinate tramite l'OR bitwise |

Per default, gli oggetti di ifstream sono aperti in lettura mentre quelli di ofstream sono aperti in scrittura. Un fstream può essere aperto sia in lettura che in scrittura.

```
fstream file("dati.txt", ios::in|ios::out);
if (file.fail()) cout << "Errore in apertura\n";
```

Apre il file "**dati.txt**" in i/o.

```
ofstream file("dati.txt", ios::app|ios::nocreate|ios::binary);
if (file.bad()) cout << "Il file non esiste\n";
```

Il file "**dati.txt**" viene aperto in modalità binaria di append alla fine.

Chiusura di un file: Il metodo **close()** chiude esplicitamente un file: viene automaticamente invocato dal distruttore dello stream.

```
class ios {
public:
   enum openmode {
   in,                  // apertura in lettura
   out,                 // apertura in scrittura
   ate,                 // spostamento a EOF dopo l'apertura
   app,                 // spostamento a EOF prima di ogni write
   trunc,               // erase file all'apertura
   binary,              // apertura in binary mode, default text mode
   };
...
}
```

member constant	opening mode
app	(**app**end) Set the stream's position indicator to the end of the stream before each output operation.
ate	(**at e**nd) Set the stream's position indicator to the end of the stream on opening.
binary	(**binary**) Consider stream as binary rather than text.
in	(**in**put) Allow input operations on the stream.
out	(**out**put) Allow output operations on the stream.
trunc	(**trunc**ate) Any current content is discarded, assuming a length of zero on opening.

Le modalità di apertura di uno stream su file possono essere combinate tramite l'OR bitwise |

Per default, gli oggetti di ifstream sono aperti in lettura mentre quelli di ofstream sono aperti in scrittura. Un fstream può essere aperto sia in lettura che in scrittura.