

# Overloading Operator

## 1 Overloading Operator <<

 Friend


## 2 Operator >>

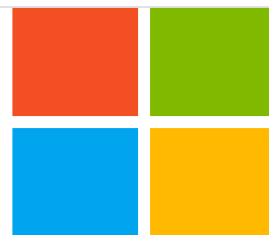
## Altri Operator

## 1 Overloading Operator <<

Overload dell'operatore << per le classi personalizzate

I flussi di input usano l'operatore di inserimento (<<) per i tipi standard. È possibile eseguire l'overload dell'operatore << per le classi personalizzate. L'esempio di funzione write ha illustrato l'uso di una struttura Date. La

 <https://learn.microsoft.com/it-it/cpp/standard-library/overloading-the-output-operator-for-your-own-classes?view=msvc-170>



tratto dalle docs di microsoft...

I flussi di input usano l'operatore di inserimento (<<) per i tipi standard. È possibile eseguire l'overload dell'operatore << per le classi personalizzate.

### Esempio:

L'esempio di funzione `write` ha illustrato l'uso di una struttura `Date`. La data rappresenta il candidato ideale per una classe C++ nella quale i membri dati (mese, giorno e anno) sono nascosti nella visualizzazione. Un flusso di output è la destinazione logica per la visualizzazione di tale struttura. Questo codice visualizza una data usando l'oggetto `cout`:

```
Date dt(1, 2, 92);  
  
cout <<dt;
```

Per fare in modo che `cout` accetti un oggetto `Date` dopo l'operatore di inserimento, eseguire l'overload dell'operatore di inserimento in modo che riconosca un oggetto `ostream` a sinistra e un oggetto `Date` a destra. La funzione dell'operatore << in overload deve quindi essere dichiarata come Friend della classe `Date` in modo che possa accedere ai dati privati all'interno di un oggetto `Date`.

```
#include <iostream>  
using namespace std;  
  
class Date {  
    int mo, da, yr;  
public:  
    Date(int m, int d, int y) {
```

```

        mo = m; da = d; yr = y;
    }
    friend ostream& operator<<(ostream& os, const Date& dt);
};

ostream& operator<<(ostream& os, const Date& dt){
    os << dt.mo << '/' << dt.da << '/' << dt.yr;
    return os;
}

int main() {
    Date dt(5, 6, 92);
    cout << dt;
}

```

Output:

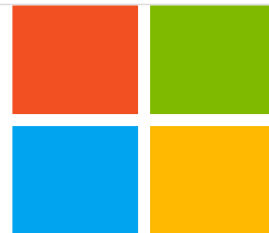
```
5/6/92
```

## Friend

### friend (C++)

In alcune circostanze, è utile per una classe concedere l'accesso a livello di membro alle funzioni che non sono membri della classe o a tutti i membri in una classe separata. Queste funzioni e classi gratuite sono

 <https://learn.microsoft.com/it-it/cpp/cpp/friend-cpp?view=msvc-170>



*tratto dalle docs di microsoft...*

In alcune circostanze, è utile per una classe concedere l'accesso a livello di membro alle funzioni che non sono membri della classe o a tutti i membri in una classe separata. Queste funzioni e classi gratuite sono note come *amici*, contrassegnate dalla `friend` parola chiave. Solo l'implementatore della classe può dichiarare i rispettivi elementi friend. Una funzione o una classe non può dichiararsi come un amico di qualsiasi classe. In una definizione di classe usare la parola chiave e il `friend` nome di una funzione non membro o di un'altra classe per concedere l'accesso ai membri privati e protetti della classe. In una definizione di modello, un parametro di tipo può essere dichiarato come `friend`.

Una `friend` funzione è una funzione che non è membro di una classe ma ha accesso ai membri privati e protetti della classe. Le funzioni friend non sono considerate membri della classe; sono normali funzioni esterne che hanno privilegi di accesso speciali. Gli amici non si trovano nell'ambito della classe e non vengono chiamati usando gli operatori di selezione membro (`e ->`) a meno che non siano membri di un'altra classe. Una `friend` funzione viene dichiarata dalla classe che concede l'accesso. La `friend` dichiarazione può essere inserita

ovunque nella dichiarazione di classe. Non è interessato dalle parole chiave del controllo di accesso.

Nell'esempio seguente viene illustrata una classe `Point` e una funzione friend `ChangePrivate`. La `friend` funzione ha accesso al membro dati privato dell'oggetto `Point` che riceve come parametro.

### Esempio:

```
#include <iostream>
using namespace std;

class Point {
    friend void ChangePrivate( Point & );
public:
    Point( void ) : m_i(0) {}
    void PrintPrivate( void ){cout << m_i << endl; }

private:
    int m_i;
};

void ChangePrivate ( Point &i ) { i.m_i++; }


int main() {
    Point sPoint;
    sPoint.PrintPrivate();
    ChangePrivate(sPoint);
    sPoint.PrintPrivate();

    // Output: 0
    //          1
}
```

## 2 Operator >>

Overload dell'operatore >> per classi personalizzate

Altre informazioni su: Overload dell'operatore >> per le classi personalizzate

 <https://learn.microsoft.com/it-it/cpp/standard-library/overloading-the-input-operator-for-your-own-classes?view=msvc-170>



*tratto dalle docs di microsoft...*

I flussi di input usano l'operatore di estrazione (`>>`) per i tipi standard. È possibile scrivere operatori di estrazione simili per i tipi personalizzati. L'esito positivo dipende dall'uso degli spazi vuoti in modo preciso.

Di seguito è riportato un esempio di un operatore di estrazione per la classe `Date` presentata in precedenza:

```
istream& operator>> (istream& is, Date& dt){
    is>> dt.mo>> dt.da>> dt.yr;
    return is;
}
```

## Altri Operator

Non esiste la ridefinizione solo degli operator di input ( `operator>>` ) e di output ( `operator<<` ), ma anche molti altri, come l' `operator==` , `operator=` , `operator>` , ecc...

Qui di seguito lascio 2 classi .hh (header file) in cui mostro alcuni esempi di funzioni operator senza implementarle.

- File ***Rational.hh***

```
#include <iostream>

#ifndef GUARDIA_CONTRO_INCLUSIONE
#define GUARDIA_CONTRO_INCLUSIONE 1

namespace Numerica{

    class Rational{
    public:
        using Integer = long; //alias

        //vari costruttori/distruttori/assegnamenti/copia
        Rational(const Integer n& = 1, const Integer d& = 1);
        Rational(const Rational&);           //costruttore di copia
        Rational(Rational&&);               //costruttore di spostamento
        Rational& operator=(const Rational&); //operatore di assegnamento per copia
        Rational& operator=(Rational&&);     //operatore di assegnamento per spostamento
        ~Rational();

        //check_inv()
        bool check_inv() const ;

        //operator
        Rational operator+() const ;
        Rational operator-() const ;

        Rational operator+(const Rational& r1, const Rational& r2);
        Rational operator-(const Rational& r1, const Rational& r2);
        Rational operator*(const Rational& r1, const Rational& r2);
        Rational operator/(const Rational& r1, const Rational& r2);

        Rational& operator+=(const Rational& r1);
        Rational& operator-=(const Rational& r1);
        Rational& operator*=(const Rational& r1);
        Rational& operator/=(const Rational& r1);
    };
}
```

```

        Rational& operator++(); //prefisso (++x)
        Rational& operator--(); //prefisso

        Rational operator++(int); //postfisso (x++)
        Rational operator--(int); //postfisso

        bool operator==(const Rational& r1, const Rational& r2) const ;
        bool operator!=(const Rational& r1, const Rational& r2) const ;
        bool operator>=(const Rational& r1, const Rational& r2) const ;
        bool operator<=(const Rational& r1, const Rational& r2) const ;
        bool operator>(const Rational& r1, const Rational& r2) const ;
        bool operator<(const Rational& r1, const Rational& r2) const ;

        //getter
        const Integer& getNum() const ;
        const Integer& getDen() const ;

    private:
        Integer num;
        Integer den;
};

} //Namespace NUMERICA

//in/out
std::istream& operator>>(std::istream& is, Rational& r);
std::ostream& operator<<(std::istream& os, const Rational& r);

#endif

```

- File **Stack.hh**

```

#ifndef GUARDIE_CONTROLLO_INCLUSIONE
#define GUARDIE_CONTROLLO_INCLUSIONE 1

template<typename T>
class Stack{
public:
    //alias
    using size_type = std::size_t;
    using value_type = T;

    //funzioni principali
    Stack(size_type capacity_ = 16);
    Stack(const Stack& x);
    Stack(Stack&& x) noexcept;
    Stack& operator=(const Stack& x);
    Stack& operator=(Stack&& x) noexcept;
    ~Stack();

    //check inv
    bool check_inv() const;

    //metodi
    bool is_empty() const;
    void push(const value_type& elem);
    void pop();

```

```

    void swap(Stack& x) noexcept;

    //getter
    const size_type& getSize() const;
    const size_type& getCapacity() const;

private:
    value_type* vec_;
    size_type capacity_;
    size_type size_;
};

//IMPLEMENTAZIONE FUNZIONI PRINCIPALI
template <typename T>
inline Stack<T>::Stack(const size_type capacity) { /*...*/ } //costruttore

template <typename T>
inline Stack<T>::Stack(const Stack& x) { /*...*/ } //costruttore di copia

template <typename T>
inline Stack<T>::Stack(Stack&& x) { /*...*/ } //costruttore per spostamento

template <typename T>
inline Stack<T>::Stack& operator=(const Stack& x) { /*...*/ } //assegnamento per copia

template <typename T>
inline Stack<T>::Stack& operator=(Stack&& x) { /*...*/ } //assegnamento per spostamento

template <typename T>
inline Stack<T>::~~Stack() { /*...*/ } //distruttore

//IMPLEMNTAZIONE CHECK INV
template <typename T>
inline bool Stack<T>::check_inv() const { /*...*/ }

//IMPLEMENTAZIONE METODI
template <typename T>
inline bool Stack<T>::is_empty() const { /*...*/ }

template <typename T>
inline void Stack<T>::push(const value_type& elem) { /*...*/ }

template <typename T>
inline void Stack<T>::pop() { /*...*/ }

template <typename T>
inline void Stack<T>::swap(Stack& x) noexcept { /*...*/ }

//IMPLEMENTAZIONE METODI GETTER
template <typename T>
inline const size_type& Stack<T>::getSize() const { /*...*/ }
inline const size_type& Stack<T>::getCapacity() const { /*...*/ }

#endif

```