

Back to information hiding



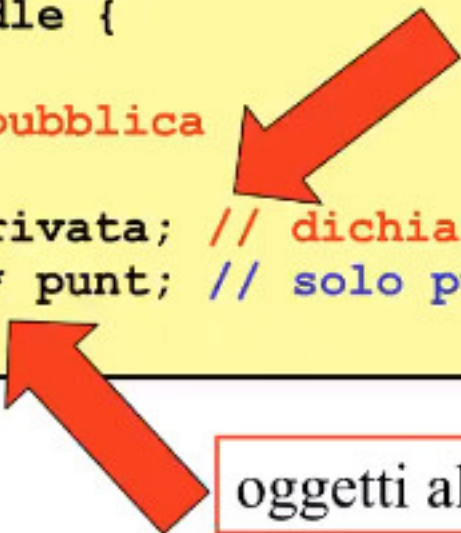
Come nascondere la parte privata

```
class C {  
public:  
    // parte pubblica  
private:  
    // parte privata  
};
```

Vogliamo nascondere fisicamente la parte privata della dichiarazione di **C** all'utente finale.



```
// file C_handle.h
class C_handle {
public:
    // parte pubblica
private:
    class C_privata; // dichiarazione incompleta
    C_privata* punt; // solo punt. e ref. per dich.incompleta
};
```



oggetti allocati sullo heap

```
// file C_handle.h
class C_handle {
public:
    // parte pubblica
private:
    class C_privata; // dichiarazione incompleta
    C_privata* punt; // solo punt. e ref. per dich.incompleta
};
```

Nell'implementazione separata di `c_handle` sarà definita la classe `C_privata` contenente la parte privata di `c`

```
// file C_handle.cpp
class C_handle::C_privata {
    // parte privata
};
```

Dichiarazioni incomplete

Una classe **C** può dichiarare ed usare puntatori e reference ad una classe **D** che viene meramente dichiarata. Si tratta di una cosiddetta *dichiarazione incompleta* della classe **D**.



Attenzione: In una classe **C** una dichiarazione che la classe **D** è amica di **C** funge anche da dichiarazione incompleta di **D**.

```
class D; // dichiar.incompleta

class C {
    D* p;                // campo dati
    D* m();              // tipo di ritorno
    void n(...,D*,...);  // parametro
    D& f(...);           // tipo di ritorno
    ...
};

class D {
    ... // definizione della classe
};
```



```
// File orario.h
#ifndef ORARIO_H
#define ORARIO_H
#include <iostream>
using std::ostream;

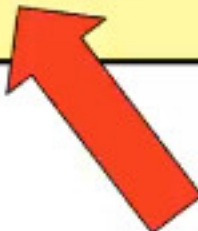

class orario {
public:
    orario(int =0,int =0,int =0);
    int Ore() const;
    int Minuti() const;
    int Secondi() const;
    void AvanzaUnOra();
private:
    class orario_rappr;
    orario_rappr* punt; // oggetti allocati sullo heap
};

ostream& operator<<(ostream&,const orario&);
#endif
```

```
// File orario.cpp
#include "orario.h"

class orario::orario_rappr {
    public:
        int sec;
}; // basta il costruttore di default standard

orario::orario(int o, int m, int s) : punt(new orario_rappr)
{
    if (o < 0 || o > 24 || m < 0 || m > 60 || s < 0 || s > 60)
        punt->sec = 0;
    else punt->sec = o*3600 + m*60 + s;
}
```



```
int orario::Ore() const
{ return punt->sec / 3600; }

int orario::Minuti() const
{ return (punt->sec - (punt->sec / 3600)*3600) / 60; }

int orario::Secondi() const
{ return punt->sec % 60; }

void orario::AvanzaUnOra()
{ punt->sec = (punt->sec + 3600) % 86400; }

ostream& operator<<(ostream& os, const orario& t)
{ return os << t.Ore() << ':' << t.Minuti() << ':'
    << t.Secondi(); }
```




Attenzione

Con questa tecnica si possono verificare problemi di interferenza!

```
orario t1(17,11,27), t2;  
cout << t1 << ' ' << t2 << endl;  
// stampa: 17:11:27 0:0:0  
t2=t1; // stesso puntatore punt in t2 e t1  
t1.AvanzaUnOra(); // interferenza  
cout << t1 << ' ' << t2 << endl;  
// stampa: 18:11:27 18:11:27
```

Sarà quindi necessario ridefinire adeguatamente assegnazione, costruttore di copia e distruttore di `orario`



DETTAGLIO CONSUMI



Sintesi Traffico Voce - Numero di Telefono

Tipologia di chiamata	Numero Chiamate	Durata	Costo IVA € (IVA inclusa)
Locale	4	0:10:37	0.00000
Nazionale	23	4:22:29	0.00000
Cellulari Nazionali	2	0:01:04	0.79198
Internazionale L	16	0:23:23	8.61132
Numeri Verdi	4	0:11:09	0.00000
Totale Traffico Voce			9.40330

Data	Ora	Numero Chiamato	Destinazione	Durata	Fascia Oraria	Costo IVA € (IVA inclusa)
Locale						
13-09-2015	18:45	0236634***	Milano	0:00:07		0.00000
26-09-2015	17:09	0236635***	Milano	0:09:11		0.00000
27-09-2015	16:56	0236635***	Milano	0:00:01		0.00000
06-10-2015	15:31	0236635***	Milano	0:01:18		0.00000

OUTPUT



```
ostream& operator<<(ostream& os, bolletta b) {  
    // NOTA BENE: b passato per valore  
    os << "TELEFONATE IN BOLLETTA" << endl;  
    int i = 1;  
    while (!b.Vuota()) {  
        os << i << ") " << b.Estrai_Una() << endl;  
        i++;  
    }  
    return os;  
}
```

COM
PORTA
MENTO
NO?



Friend function

From Wikipedia, the free encyclopedia

In [object-oriented programming](#), a **friend function** that is a "friend" of a given [class](#) is allowed access to `private` and `protected` [data](#) in that class that it would not normally be able to as if the data was `public`.^[1] Normally, a [function](#) that is defined outside of a class cannot access such information. Declaring a function a **friend** of a class allows this, in languages where the concept is supported.

A friend function is declared by the class that is granting access, explicitly stating what function from a class is allowed access. A similar concept is that of [friend class](#).

Friends should be used with caution. Too many functions or external classes declared as friends of a class with protected or private data may lessen the value of [encapsulation](#) of separate classes in object-oriented programming and may indicate a problem in the overall architecture design. Generally though, friend functions are a good thing for encapsulation, as you can keep data of a class private from all except those who you explicitly state need it, but this does mean your classes will become tightly coupled.

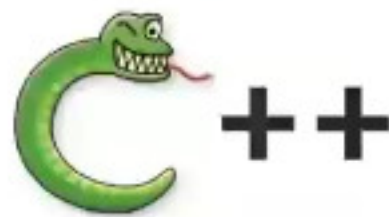


```
// file bolletta.h
class bolletta {
    ...
    // funzione esterna dichiarata friend
    friend ostream& operator<<(ostream&, const bolletta&) ;
    ...
};
```

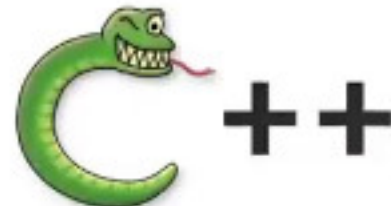


```
// file bolletta.h
class bolletta {
    ...
    // funzione esterna dichiarata friend
    friend ostream& operator<<(ostream&, const bolletta&);
    ...
};
```

```
// nel file bolletta.cpp
ostream& operator<<(ostream& os, const bolletta& b) {
    os << "TELEFONATE IN BOLLETTA" << endl;
    bolletta::nodo* p = b.first; // per "amicizia"!
    int i = 1;
    while (p) {
        os << i++ << " ) " << p->info << endl;
        p = p->next;
    }
    return os;
}
```



Accesso per classi annidate



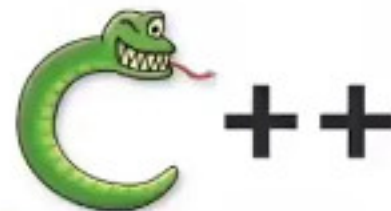
The C++ Standard (2003) says in §11.8/1 [class.access.nest],

The members of a nested class have no special access to members of an enclosing class, nor to classes or functions that have granted friendship to an enclosing class; the usual access rules (clause 11) shall be obeyed. The members of an enclosing class have no special access to members of a nested class; the usual access rules (clause 11) shall be obeyed.

Example from the Standard itself:

```
class E
{
    int x;
    class B { };
    class I
    {
        B b; // error: E::B is private
        int y;
        void f(E* p, int i)
        {
            p->x = i; // error: E::x is private
        }
    };
    int g(I* p)
    {
        return p->y; // error: I::y is private
    }
};
```

§11.8/1 in C++98 states:



The members of a nested class **have no special access** to members of an enclosing class, nor to classes or functions that have granted friendship to an enclosing class; the usual access rules shall be obeyed.

§11.8/1 in N1804(after TR1) states:

A nested class is a member and as such **has the same access rights** as any other member.



da C++11



standard §11.7.1

"A nested class is a member and as such has the same access rights as any other member. The members of an enclosing class have no special access to members of a nested class; the usual access rules shall be obeyed"

and the usual access rules specify that:

"A member of a class can also access all the names to which the class has access..."

```
class E {  
    int x;  
    class B { };  
  
    class I {  
        B b;  
        int y;  
        void f(E* p, int i) {  
            p->x = i;  
        }  
    };  
  
    int g(I* p) {  
        return p->y;  
    }  
};
```

// OK: E::I can access E::B



// OK: E::I can access E::x



// error: I::y is private



Nota Bene: (come nella vita) la relazione di amicizia non è simmetrica e non è transitiva (vale per amicizia tra classi).



Iteratori: Funzionalità per scorrere ed accedere agli elementi di una collezione



Iteratori: Funzionalità per scorrere ed accedere agli elementi di una collezione

Iterator

From Wikipedia, the free encyclopedia

In [object-oriented computer programming](#), an **iterator** is an [object](#) that enables a programmer to traverse a [container](#), particularly [lists](#).^{[1][2][3]} Various types of iterators are often provided via a container's interface. Though the interface and semantics of a given iterator are fixed, iterators are often implemented in terms of the structures underlying a container implementation and are **often tightly coupled to the container** to enable the operational semantics of the iterator. Note that an iterator performs traversal and also gives access to data elements in a container, but does not perform iteration (i.e., not without some significant liberty taken with that concept or with trivial use of the terminology).



```
class contenitore {  
private:  
    class nodo {  
public: // per convenienza nell'esempio  
        int info;  
        nodo* next;  
        nodo(int x, nodo* p): info(x), next(p) {}  
    };  
    nodo* first; // puntatore al primo nodo della lista  
  
public:  
    contenitore(): first(0) {}  
    void aggiungi_in_testa(int x) {first = new nodo(x,first);}  
}
```



Classe `iteratore` i cui oggetti rappresentano degli indici ai nodi degli oggetti della classe `contenitore`.

```
class iteratore {
private:
    contenitore::nodo* punt; // nodo puntato dall'iteratore
public:
    bool operator==(const iteratore& i) const {
        return punt == i.punt;
    }
    bool operator!=(const iteratore& i) const {
        return punt != i.punt;
    }
    iteratore& operator++() { // operator++ prefisso
        if (punt) punt = punt->next; return *this;
    }
    // se it punta all'ultimo nodo, da ++it non si torna indietro
    // nessun costruttore per il momento
};
```