

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
};
```

**Problema:** nodo ed i suoi membri non sono accessibili dall'esterno


```
class contenitore {  
    // friend class iteratore; // non necessaria da C++03  
private:  
    class nodo {  
        ...  
    };  
    nodo* first;  
public:  
    class iteratore { // classe annidata nella parte pubblica  
    private:  
        contenitore::nodo* punt;  
        ...  
    };  
    contenitore();  
    void aggiungi_nodo(int);  
}
```

**!CAUTION!**



**EVERYWHERE**

```
class contenitore {  
  
private:  
    class nodo { ... };  
    nodo* first;  
public:  
    class iteratore {  
        friend class contenitore; // dichiarazione di amicizia  
        ...  
    }; // Attenzione, va definita prima dei metodi che la usano  
    ...  
    iteratore begin() const; // "costruttore di iteratore"  
    iteratore end() const; // "costruttore di iteratore"  
    // operatore di subscripting  
    int& operator[] (const iteratore&) const;  
};
```



**!CAUTION!**



**EVERYWHERE**

```
contenitore::iteratore contenitore::begin() const {  
    iteratore aux;    // costruttore di default standard  
    aux.punt = first; // per amicizia ho accesso a punt  
    return aux;  
}  
  
contenitore::iteratore contenitore::end() const {  
    iteratore aux;  
    aux.punt = 0; // per amicizia  
    return aux;  
}  
  
int& contenitore::operator[](const contenitore::iteratore& it) const {  
    return it.punt->info; // per amicizia, nessun controllo su it.punt  
}
```



Possiamo ora utilizzare gli iteratori della classe contenitore come nel seguente esempio di funzione esterna:

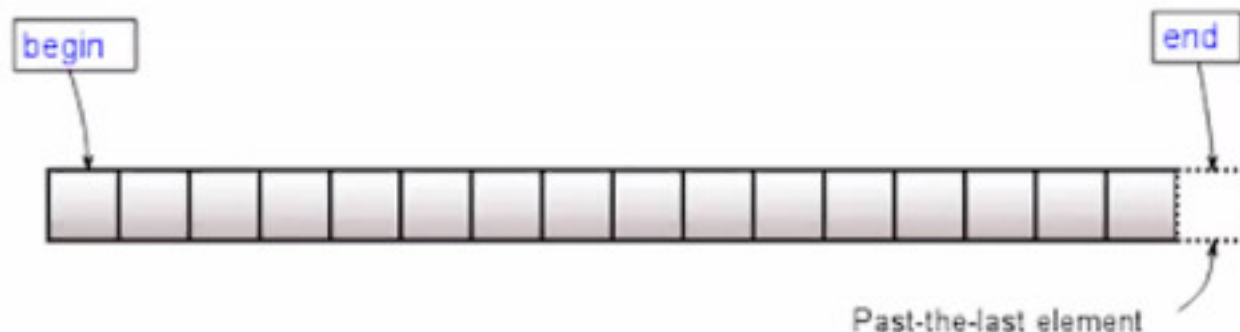
```
int somma_elementi(const contenitore& c) {  
    int s=0;  
    for(contenitore::iteratore it=c.begin(); it!=c.end(); ++it)  
        s += c[it];  
    return s;  
}
```

## Dichiarazione della classe `iteratore`:

```
class iteratore {  
    friend class bolletta;  
public:  
    bool operator==(const iteratore&) const;  
    bool operator!=(const iteratore&) const;  
    iteratore& operator++();           // ++ prefisso  
    iteratore operator++(int);        // ++ postfisso  
private:  
    bolletta::nodo* punt;  
};  
// notare che non ci sono costruttori espliciti
```

Metodi `begin()` ed `end()` e l'overloading dell'operatore di indicizzazione `[]`.

```
class bolletta {  
public:  
    ...  
    iteratore begin() const;  
    iteratore end() const;  
    telefonata& operator[](const iteratore&) const;  
    // tipo di ritorno per riferimento  
    ...  
private:  
    ...  
};
```



```
// bolletta.h
#ifndef BOLLETTA_H
#define BOLLETTA_H
#include "telefonata.h"

class bolletta {
private:
    class nodo {
    public:
        nodo();
        nodo(int x, nodo* p): info(x), next(p) {}
        telefonata info;
        nodo* next;
        ~nodo(); // distruttore "ricorsivo"
    };
    nodo* first; // puntatore al primo nodo della lista
    static nodo* copia(nodo*);
    static void distruggi(nodo*);
};

// continua ...
```



```
// ... continuazione
```

```
public:
```

```
    class iteratore {
```

```
        friend class bolletta;
```

```
    private:
```

```
        bolletta::nodo* punt; // nodo puntato dall'iteratore
```

```
    public:
```

```
        bool operator==(const iteratore&) const;
```

```
        bool operator!=(const iteratore&) const;
```

```
        iteratore& operator++(); // operator++ prefisso
```

```
        iteratore operator++(int); // operator++ postfisso
```

```
}; // end classe iteratore
```

```
bolletta();
```

```
~bolletta(); // distruzione profonda
```

```
bolletta(const bolletta&); // copia profonda
```

```
bolletta& operator=(const bolletta&); // assegnazione profonda
```

```
bool Vuota() const;
```

```
void Aggiungi_Telefonata(const telefonata& t);
```

```
void Togli_Telefonata(const telefonata& t)
```

```
    telefonata Estrai_Una();
```

```
    // metodi che usano iteratore
```

```
    iteratore begin() const;
```

```
    iteratore end() const;
```

```
    telefonata& operator[](const iteratore&) const;
```

```
};
```

```
#endif
```

```
// bolletta.cpp
#include "bolletta.h"

bool bolletta::iteratore::operator==(const iteratore&) const {
    return punt == i.punt;
}

bool bolletta::iteratore::operator!=(const iteratore&) const {
    return punt != i.punt;
}

bolletta::iteratore& bolletta::iteratore::operator++() { // prefisso
    if (punt) punt = punt->next; //side-effect
    return *this;
} // NB: se punt==0 non fa nulla

bolletta::iteratore bolletta::iteratore::operator++(int) { // postfisso
    iteratore aux = *this;
    if (punt) punt = punt->next; //side-effect
    return aux;
}
```

```
// continuazione file bolletta.cpp
```

```
bolletta::iteratore bolletta::begin() const {  
    bolletta::iteratore aux;  
    aux.punt = first; // amicizia  
    return aux;  
}
```

```
bolletta::iteratore bolletta::end() const {  
    bolletta::iteratore aux;  
    aux.punt = nullptr; // amicizia  
    return aux;  
}
```

```
telefonata& bolletta::operator[] (const bolletta::iteratore& it) const {  
    return (it.punt)->info; // amicizia  
    // NB: nessun controllo it.punt != 0  
}
```





L'utente esterno può calcolare la somma delle durate delle telefonate di una bolletta con la seguente funzione:

```
orario Somma_Durate(const bolletta& b) { // per riferimento
    orario durata;
    for(bolletta::iteratore it = b.begin(); it != b.end(); ++it)
        durata = durata + (b[it].Fine() - b[it].Inizio());
    return durata;
}
```

## Esercizio:

Ridefinire l'operatore "**telefonata& operator\*()**" come metodo della classe **iteratore** cosicchè la funzione **Somma\_Durate** possa essere scritta nel seguente modo:

```
orario Somma_Durate(const bolletta& b) { // per riferimento
    orario durata;
    for (bolletta::iteratore it = b.begin(); it != b.end(); ++it)
        durata = durata + ((*it).Fine() - (*it).Inizio());
    return durata;
}
```



## Overloading operator->

L'operatore -> di selezione di membro tramite puntatore può essere ridefinito internamente *come operatore unario postfixo*.

Tipicamente ritorna un puntatore ad una classe (oppure un oggetto di una classe per cui è stato ridefinito ->).

Definire l'overloading di -> come metodo di iteratore in modo tale che Somma\_Durate() possa essere scritta nel seguente modo:

```
orario Somma_Durate(const bolletta& b) { // per riferimento
    orario durata;
    for (bolletta::iteratore it = b.begin(); it != b.end(); ++it)
        durata = durata + (it->Fine() - it->Inizio());
    return durata;
}
```

```

public:
    class iteratore {
    friend class bolletta;
    private:
        bolletta::nodo* punt; // nodo puntato dall'iteratore
    public:
        bool operator==(const iteratore&) const;
        bool operator!=(const iteratore&) const;
        iteratore& operator++(); // operator++ prefisso
        iteratore operator++(int); // operator++ postfisso
        telefonata* operator->() const {return &(punt->info);}
        telefonata& operator*() const {return punt->info;}
    };
    bolletta();
    ~bolletta(); // distruzione profonda
    bolletta(const bolletta&); // copia profonda
    bolletta& operator=(const bolletta&); // assegnazione profonda
    bool Vuota() const;
    void Aggiungi_Telefonata(const telefonata& t);
    void Togli_Telefonata(const telefonata& t)
    telefonata Estrai_Una();
    // metodi che usano iteratore
    iteratore begin() const;
    iteratore end() const;
    telefonata& operator[](const iteratore&) const;
};

```

```
public:
    class iteratore {
    friend class bolletta;
    private:
        bolletta::nodo* punt; // nodo puntato dall'iteratore
    public:
        bool operator==(const iteratore&) const;
        bool operator!=(const iteratore&) const;
        iteratore& operator++(); // operator++ prefisso
        iteratore operator++(int); // operator++ postfisso
        telefonata* operator->() const {return &(punt->info);}
        telefonata& operator*() const {return punt->info;}
```

**Question: perché questi metodi  
const sono compatibili con tipi di  
ritorno non const?**

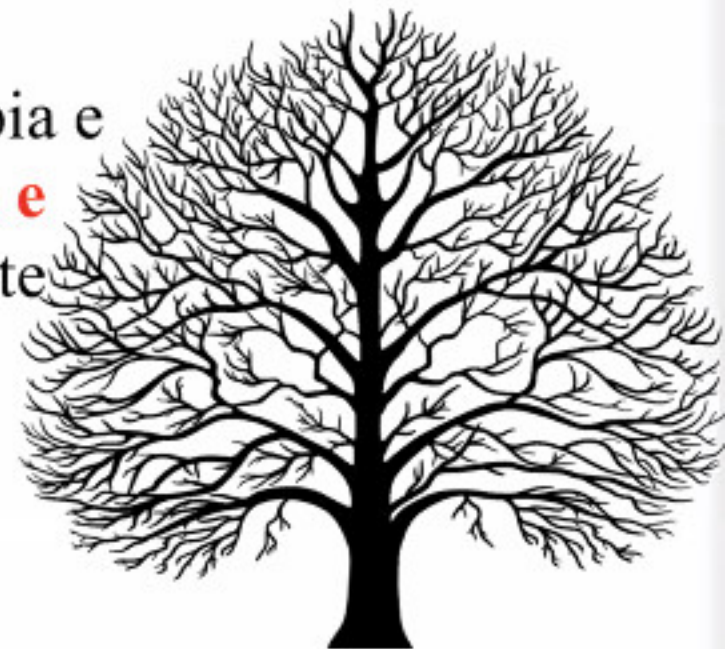
```
};
```



## ESERCIZI DELLA PROSSIMA LEZIONE

```
class Nodo {  
private:  
    Nodo(char c='*', Nodo* s=0, Nodo* d=0): info(c), sx(s), dx(d) {}  
    char info;  
    Nodo* sx;  
    Nodo* dx;  
};  
class Tree {  
public:  
    Tree(): root(0) {}  
    Tree(const Tree&); // dichiarazione costruttore di copia  
private:  
    Nodo* root;  
};
```

Gli oggetti della classe Tree rappresentano alberi binari ricorsivamente definiti di char. Si ridefiniscano assegnazione, costruttore di copia e distruttore di Tree come **assegnazione, copia e distruzione profonda**. Scrivere esplicitamente eventuali dichiarazioni friend che dovessero essere richieste da tale definizione.



Definire una classe **Vettore** i cui oggetti rappresentano array di interi. Vettore deve includere un costruttore di default, una operazione di concatenazione che restituisce un nuovo vettore  $v1+v2$ , una operazione di append  $v1.append(v2)$ , l'overloading dell'uguaglianza, dell'operatore di output e dell'operatore di indicizzazione. Deve inoltre includere il costruttore di copia profonda, l'assegnazione profonda e la distruzione profonda.

