Classi e Oggetti

Una *classe* definisce un nuovo tipo di dato che può essere utilizzato per la creazione di *oggetti*. Una classe viene definita mediante la keyword **class**. La dichiarazione di una classe prevede la definizione sia dei dati che delle funzioni che appartengono alla classe: i dati sono detti *data members*, le funzioni *member functions* o *metodi* della classe.

I membri di una classe possono essere privati, protetti o pubblici; per default tutti gli elementi definiti in una classe sono privati, cioè accessibili solo dai membri della classe stessa.
Per rendere accessibili dati e funzioni di una classe anche ad altre parti del programma, questi devono essere dichiarati dopo la keyword **public**. In questo modo la dichiarazione di una classe assume la seguente forma standard:

```
class nome_classe {
    dati e metodi privati

public:
    dati e metodi pubblici
};
```

Esempio: classe queue (coda di 100 elementi interi)

```
class queue {
private:        // superfluo
    int q[100];
    int sloc, rloc;

public:
    void init();
    void qput(int i);
    int qget();
};
```

Una volta dichiarata una classe, è possibile creare un oggetto di quel tipo utilizzando il nome della classe:

```
queue c1;
c1.init();      // invocazione di un metodo
```

Le member functions della classe vengono implementate mediante una sintassi che indica il nome della classe e l'operatore di *scope resolution*:

```
void queue::init() {
    rloc = sloc = 0;
    cout << "Coda inizializzata"
}

void queue::qput(int i) {
    if (sloc==100) { cout << "coda piena"; return; }
    sloc++;
    q[sloc] = i;
}

int queue::qget() {
    rloc++;
    return q[rloc];
}
```

I dati della classe sono privati. Un tentativo di accesso causa un errore di compilazione.

```
queue a;
a.rloc = 0;  // errore!
```

I metodi di una classe non sono definiti nel namespace globale, quindi possono essere invocati solo in associazione ad un oggetto della classe usando l'operatore punto o →.
Per contro, possono esistere metodi di classi diverse aventi lo stesso nome e gli stessi parametri senza incorrere in ambiguità, proprio per effetto del fatto che i nomi delle member functions non vengono ricercati nel namespace globale.

La definizione di una classe soddisfa due concetti di base del paradigma OOP:

ENCAPSULATION = organizzare gli oggetti in modo tale che essi siano descritti insieme alle relative operazioni
INFORMATION HIDING = nascondere tutti i dettagli di un oggetto che non concorrono alla sua natura essenziale visibile dall'esterno

Una classe può essere vista come una struct con tutti i dati privati. L'atto di portare le funzioni dentro le strutture rappresenta la base di ciò che il C++ aggiunge al C, e introduce un modo diverso di pensare alle strutture: come concetti.
In C, una struct è un insieme di dati aggregati, che vengono allocati, copiati e

passati come parametri formali tutti insieme. Il C è un linguaggio fortemente strutturato, cioè fornisce un supporto molto efficiente all'utilizzo di dati aggregati.
Ma è difficile pensare alle struct come a qualcosa di diverso da un utile tool per il programmatore, perché le funzioni che operano sulla struttura sono definite ed implementate altrove.
Invece, un costrutto che incorpora dati e funzioni diventa una nuova creatura, capace di descrivere caratteristiche e comportamenti.
Il concetto di oggetto, alla base di OOP, è quello di entità dai confini definiti, che può vivere di vita propria, ricordare ed agire.

La dichiarazione di una classe definisce un nuovo data type: così come il data type predefinito float ha esponente, mantissa e bit di segno, e prevede una serie di operazioni, un oggetto definito dall'utente prevede dati ed operazioni.
Inoltre, il meccanismo di type checking del C++ fa sì che se una funzione si aspetta un determinato oggetto come parametro, l'invocazione di quella funzione passando qualsiasi altro data type causa un errore di compilazione.
Ovviamente nel caso di funzioni aventi oggetti come parametri non esistono meccanismi di conversione automatica di tipo.

Anche se le classi si comportano come i tipi predefiniti, vengono definiti *abstract data types*, perché permettono al programmatore di astrarre un concetto dal problem space.

Costruttori e Distruttori

Nell'esempio precedente, il metodo init() provvede all'inizializzazione dei dati della classe queue. L'uso di funzioni di questo tipo è inelegante e *error prone*, perché l'inizializzazione non è obbligatoria e il programmatore può facilmente dimenticarla.
Un migliore approccio consiste nel permettere al programmatore di dichiarare una funzione con l'esplicito scopo di inizializzare un oggetto. Una funzione di questo tipo viene detta *costruttore*, ed assume lo stesso nome della classe.

Complementare al costruttore è il *distruttore* della classe, utilizzato principalmente per compiere operazioni preliminari alla distruzione di oggetti (compiuta nel momento in cui vanno out of scope).
Un distruttore assume lo stesso nome della classe, preceduto dal simbolo **~**.

```
class queue {
private:
    int q[100];
    int sloc, rloc;

public:
    queue();        // costruttore
    ~queue();       // distruttore
    void qput(int i);
    int qget();
};
```

L'implementazione di costruttore e distruttore della classe queue assume la forma:

```
queue::queue() {
    sloc = rloc = 0;   // inizializzazione data members
    cout << "coda inizializzata" << endl;
}

queue::~queue() {
    cout << "coda distrutta" << endl;
}
```

Notare che la definizione di costruttore e distruttore di una classe non è obbligatoria. Se vengono omessi, il programma crea oggetti quando vengono dichiarati e li distrugge quando vanno out of scope senza eseguire operazioni particolari.
Tuttavia è buona norma definire sempre costruttore e distruttore di una classe.

Invocazione di costruttore e distruttore

```
int main()
{
  queue a, b;  // Creazione di due oggetti queue
               // viene invocato il costruttore queue()
  a.qput(10);
  b.qput(19);
  a.qput(20);
  b.qput(1);

  cout << a.qget() << " " << a.qget() << " ";       // 10 20
  cout << b.qget() << " " << b.qget() << endl;      // 19  1

  return 0;
}              // viene invocato il distruttore ~queue()
```

Nell'esempio precedente costruttore e distruttore sono stati invocati per effetto dei meccanismi di gestione statica della memoria.
Costruttori e distruttori vengono invocati anche facendo uso di allocazione dinamica:

```
int main()
{
  queue *a = new queue,    // Creazione di due oggetti queue
        *b = new queue;    // viene invocato il costruttore queue()

  a->qput(10);
  b->qput(19);
  a->qput(20);
  b->qput(1);

  cout << a->qget() << " " << a->qget() << " ";
  cout << b->qget() << " " << b->qget() << endl;

  delete a;          // viene invocato il distruttore ~queue()
  delete b;

  return 0;
}
```

Costruttori con parametri

Un costruttore può avere dei parametri. Questo permette di assegnare dei valori ai data members nel momento in cui un oggetto viene creato.

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

// Dichiarazione della classe queue.
class queue {
private:
  int q[100];
  int sloc, rloc;
  int who;          // contiene il numero identificativo della coda

public:
  queue(int id);    // costruttore con parametri
  ~queue();         // distruttore
  void qput(int i);
  int qget();
};
```

```
// costruttore con parametri
queue::queue(int id)
{
  sloc = rloc = 0;
  who = id;
  cout << "Coda " << who << " inizializzata.\n";
}

// distruttore
queue::~queue()
{
  cout << "Coda " << who << " distrutta.\n";
}

// Mette un intero  nella coda.
void queue::qput(int i)
{
  if(sloc==100) {
    cout << "La coda è piena.\n";
    return;
  }
  sloc++;
  q[sloc] = i;
}
```

```
// Ottiene un intero dalla coda.
int queue::qget()
{
  if(rloc == sloc) {
    cout << "Underflow della coda.\n";
    return 0;
  }
  rloc++;
  return q[rloc];
}

int main()
{
  queue a(1), b(2);  // Creazione di due oggetti queue

  a.qput(10);
  b.qput(19);
  a.qput(20);
  b.qput(1);

  cout << a.qget() << " " << a.qget() << " ";
  cout << b.qget() << " " << b.qget() << endl;

  return 0;
}
```

Nell'esempio precedente, il costruttore ha un solo parametro; in generale i costruttori possono avere più parametri.
Notare che in una classe possono esistere più costruttori (con parametri diversi), che vengono distinti dal compilatore per mezzo dei meccanismi di overloading.

A differenza dei costruttori, i distruttori non possono avere parametri.
Il motivo è semplice: non ha senso passare argomenti ad un oggetto che viene distrutto.
In ogni classe esiste un solo distruttore.

Nel caso in cui un costruttore ha un solo parametro, come nell'esempio precedente, è possibile invocarlo con una sintassi alternativa:

```
int main()
{
  queue a = 1,     // equivale a queue a(1)
        b = 2;     // equivale a queue b(2)
  …
  return 0;
}
```

Esempio: Libreria Stash

Ritornamo all'esempio CStash visto in precedenza, e riscriviamolo in C++ facendo uso di una classe.

```
class Stash {
private:
    int size;      // Dimensione di ogni elemento (in bytes)
    int quantity;  // Numero di elementi che può contenere
    int next;      // Prossimo elemento libero
    // Array di bytes allocato dinamicamente:
    unsigned char* storage;

public:
    void initialize(int size);
    void cleanup();
    int add(const void* element);
    void* fetch(int index);
    int count();
    void inflate(int increase);
};
```

Notare che:

- Nella versione C vista in precedenza tutte le funzioni che operano su una struttura CStash prevedevano un puntatore alla struttura stessa come primo parametro formale. Nella versione C++ non è più necessario: tutte le member functions di una classe operano su un'istanza di quella classe.
- Inoltre è scomparso il typedef iniziale perché in C++ la dichiarazione di una classe è una definizione di un nuovo data type.
- Tutte le member functions operano sicuramente su oggetti del tipo Stash, quindi non è necessario utilizzare nomi di funzione che fanno riferimento al data type (come CStash_initialize, CStash_add, etc.) → migliora la leggibilità del programma.

Implementazione delle member functions:

```
void Stash::initialize(int sz) {
    size = sz;
    quantity = 0;
    storage = 0;
    next = 0;
}


int Stash::add(const void* element) {
    if(next >= quantity)
        inflate(increment);  //Se necessario amplia la struttura

    // Copia l'elemento nello storage (byte per byte),
    // iniziando alla prima posizione disponibile
    int startBytes = next * size;
    unsigned char* e = (unsigned char*)element;
    for (int i = 0; i < size; i++)
        storage[startBytes + i] = e[i];
    next++;
    return (next - 1); // Ritorna l'indice
}
```

```
void Stash::inflate(int increase) {
    int newQuantity = quantity + increase;
    int newBytes = newQuantity * size;
    int oldBytes = quantity * size;

    // allocazione nuovo array
    unsigned char* b = new unsigned char[newBytes];

    // Copia da vecchio a nuovo array
    for(int i = 0; i < oldBytes; i++)
        b[i] = storage[i];

    delete [](storage); // Dealloca vecchio array
    storage = b; // Storage punta alla nuova locazione
    quantity = newQuantity;
}
```

… simile per le altre member functions.

Utilizzo della classe Stash

```
#include "CppLib.h"
using namespace std;

int main() {
    Stash intStash;
    int i;

    // Inizializzazione: specifichiamo che si tratta di una struttura che
    // contiene degli interi
    intStash.initialize(sizeof(int));
    for (i = 0; i < 150; i++)
        intStash.add(&i);      // ad ogni locazione assegniamo
                               // un numero progressivo

    for(i = 0; i < intStash.count(); i++)
        cout << "Elemento " << i << " = "
        << *(int*)intStash.fetch(i)
        << endl;

    // fine: deallocazione memoria
    intStash.cleanup();
    return 0;
}
```

In questa versione la classe Stash non prevede costruttore e distruttore.

Analogamente a quanto visto in precedenza:

- L'inizializzazione può essere demandata al costruttore: il metodo initialize può essere omesso.
- La liberazione della memoria può essere demandata al distruttore: il metodo cleanup può essere omesso.

Inoltre, si può notare che il metodo inflate viene invocato solo nel metodo add della classe. Possiamo quindi dire che inflate non rappresenta una parte dell'interfaccia della classe Stash, ma una sua implementazione interna.
E' quindi preferibile dichiarare inflate come metodo privato: in questo modo siamo liberi di modificare in tempi successivi l'implementazione della gestione della memoria.

Con queste considerazioni, la classe Stash diventa:

```
class Stash {
private:
    int size;      // Dimensione di ogni elemento (in bytes)
    int quantity;  // Numero di elementi che può contenere
    int next;      // Prossimo elemento libero
    // Array di bytes allocato dinamicamente:
    unsigned char* storage;
    void inflate(int increase);   // metodo privato

public:
    Stash(int sz);          // Costruttore
    ~Stash();               // Distruttore
    //void initialize(int sz);      <-- non serve più
    //void cleanup();               <-- non serve più
    int add(const void* element);
    void* fetch(int index);
    int count();
};
```

```
Stash::Stash(int sz) {
    size = sz;
    quantity = 0;
    storage = 0;
    next = 0;
}

Stash::~Stash() {
    if(storage != 0)
        delete[] storage;
}
```

... gli altri metodi della classe restano invariati.

Inizializzazione data members

Nell'esempio queue il costruttore provvede all'assegnamento dei data members sloc e rloc:

```
queue::queue(int id)
{
  sloc = rloc = 0;          // assegnamento data members
  who = id;
  cout << "Coda " << who << " inizializzata.\n";
}
```

Nei costruttori di una classe è possibile farlo anche con una sintassi alternativa, detta *inizializzazione dei data members*:

```
queue::queue(int id) :
  sloc(0), rloc(0)          // inizializzazione data members
{
  who = id;
  cout << "Coda " << who << " inizializzata.\n";
}
```

Le due modalità sono entrambe valide, ma in termini di efficienza è preferibile usare l'inizializzazione. Infatti la creazione di un oggetto tramite un costruttore viene effettuata in due fasi dal sistema:

- Prima vengono inizializzati tutti i data members (in maniera implicita se non si è specificata l'inizializzazione)
- Poi viene eseguito il codice del costruttore.

Quindi l'assegnamento del data member viene in realtà effettuato prima creandone un'istanza con il valore di default, poi assegnandone il valore desiderato → sono due istruzioni: se il data member è un oggetto complesso potrebbero essere anche onerose.
Viceversa, se si usa l'inizializzazione dei data members, la creazione dell'istanza viene effettuata assegnando direttamente il valore desiderato → è una sola istruzione.

➢ Scrivere il costruttore della classe Stash usando l'inizializzazione dei data members.

Il puntatore this

Una classe C++ ha member functions che operano direttamente su un'istanza della classe stessa → nelle member functions non è necessario passare un puntatore alla classe, contrariamente a quanto si è costretti a fare con le librerie C (si veda l'esempio CStash).
Tuttavia questo puntatore è implicitamente passato alle member functions di una classe. In altri termini, possiamo dire che ogni chiamata ad un metodo di una classe porta con sé il passaggio automatico di un puntatore, denominato **this**, all'oggetto sul quale è chiamato.
Il puntatore this è un parametro implicito per tutte le member functions di una classe, dunque all'interno di esse è possibile utilizzarlo per riferirsi all'oggetto chiamante.
La keyword this, utilizzabile all'interno dei metodi di una classe, produce un puntatore contenente l'indirizzo dell'oggetto nel quale è invocato.

Esempio

```
class cl {
  int i;
public:
  void load_i(int val);
  int get_i();
};

void cl::load_i(int val) {
    this->i = val;     // analogo a i = val
}

int cl::get_i() {
    return this->i;    // analogo a return i
}


int main()
{
    cl o;
    o.load_i(100);
    cout << o.get_i();
    return 0;
}
```

Funzioni inline all'interno di una classe

E' possibile dichiarare una funzione inline inserendo direttamente il codice che implementa una member function all'interno della dichiarazione di classe. Qualsiasi funzione definita in una dichiarazione di classe viene automaticamente considerata inline dal compilatore, senza specificare la keyword inline.

Nell'esempio precedente, load_i e get_i sono funzioni molto semplici che può essere conveniente definire inline:

```
class cl {
  int i;
public:
  void load_i(int val) { this->i = val; }    // analogo a i = val
  int get_i() { return this->i; }            // analogo a return i
};
```

in questo modo load_i e get_i sono definite all'interno della dichiarazione della classe cl, quindi sono automaticamente considerate inline dal compilatore.

Dati statici di una classe

A volte è necessario che tutti gli oggetti che istanziano una classe accedano ad un dato globale. Ad esempio, potremmo gestire un contatore che tiene traccia di quanti oggetti di una classe sono stati creati.

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

int count = 0;     // variabile globale

// Dichiarazione della classe queue.
class queue {
private:
  int q[100];
  int sloc, rloc;
  int who;

public:
  queue(int id);
  ~queue();
  void qput(int i);
  int qget();
};
```

```
queue::queue(int id)
{
  sloc = rloc = 0;
  who = id;
  count++;          // incrementa il contatore
  cout << "Coda " << who << " inizializzata.\n";
  cout << "Esistono " << count << " code.\n";
}

queue::~queue()
{
  count--;          // decrementa il contatore
  cout << "Coda " << who << " distrutta.\n";
  cout << "Esistono " << count << " code.\n";
}

int main()
{
  queue a(1), b(2);

  ...

  return 0;
}
```

Possiamo notare che count, nonostante sia un dato globale, è utilizzato solamente dai membri della classe.
In casi come questo, anziché utilizzare un dato globale, è preferibile ricorrere ai dati statici di una classe. Un **dato statico** di una classe agisce come un dato globale che appartiene alla classe nel quale è definito. Viene dichiarato anteponento la keyword **static** alla dichiarazione del dato nella dichiarazione della classe.
A differenza degli altri data members, nel sistema esiste una sola occorrenza di un dato statico per ogni classe: un dato statico è un dato singolo e condiviso da tutti gli oggetti che istanziano la classe.

Ci sono due vantaggi nell'utilizzo di data members statici invece di dati globali:

1. Il nome di un membro statico di una classe non viene ricercato nel namespace globale, quindi si riduce la possibilità di conflitti sui nomi.
2. E' possibile sfruttare le restrizioni di accesso ai dati: i membri statici possono essere privati.

```
// Dichiarazione della classe queue.
class queue {
private:
  int q[100];
  int sloc, rloc;
  int who;
  static int count;        // dichiarazione del dato statico

public:
  queue(int id);
  ~queue();
  void qput(int i);
  int qget();
};

// definizione del dato statico
int queue::count = 0;

queue::queue(int id)
{
  sloc = rloc = 0;
  who = id;
  count++;          // incrementa il contatore
  cout << "Coda " << who << " inizializzata.\n";
  cout << "Esistono " << count << " code.\n";
}
```

```
queue::~queue()
{
  count--;          // decrementa il contatore
  cout << "Coda " << who << " distrutta.\n";
  cout << "Esistono " << count << " code.\n";
}

int main()
{
  queue a(1), b(2);

  ...

  return 0;
}
```

Notare che i membri statici di una classe non possono essere inizializzati al momento della dichiarazione: è necessario provvedere alla loro definizione. Ciò viene fatto con una istruzione posta al di fuori dello scope globale:

```
// definizione del dato statico
int queue::count = 0;
```

Apparentemente questa istruzione sembra una violazione delle regole di restrizione di accesso ai dati privati della classe, ma in realtà non è così. Si tratta invece di un meccanismo per consentire l'inizializzazione dei membri statici di classe, che altrimenti sarebbe impossibile.

E’ possibile definire anche delle costanti come membri statici di classe. In questo caso è possibile assegnarne il valore al momento della dichiarazione.

```
// Dichiarazione della classe queue.
class queue {
private:
  int q[100];
  int sloc, rloc;
  int who;
  static int count;                // dato statico variabile
  static const int qlen = 100;     // dato statico costante

public:
  queue(int id);
  ~queue();
  void qput(int i);
  int qget();
};

// definizione del dato statico
int queue::count = 0;
```

Se un dato statico è definito come pubblico, è possibile accedervi anche dall'esterno. Per farlo però è necessario specificare lo scope della classe, perché il dato statico non è definito nel namespace globale.

```
class queue {
private:
  int q[100];
  int sloc, rloc;
  int who;

public:
  queue(int id);
  ~queue();
  void qput(int i);
  int qget();
  static int count;                    // dato statico variabile (pubblico)
};

int main()
{
  queue a(1), b(2);
  cout << queue::count << endl;          // accesso al dato statico
  return 0;
}
```

Funzioni statiche di una classe

Analogamente ai data members, per le classi è possibile definire anche *static member functions*. Vengono dichiarate utilizzando la keyword **static** nella dichiarazione della funzione. Sono particolarmente utili per maneggiare dati statici.

```
class queue {
private:
  int q[100];
  int sloc, rloc;
  int who;
  static int count;

public:
  queue(int id);
  ~queue();
  void qput(int i);
  int qget();
  static int scrivi();              // static member function
};
```

L'implementazione delle funzioni statiche non deve ripetere la keyword static.
A differenza degli altri metodi, nel sistema esiste una sola occorrenza di una funzione statica per ogni classe.
Di conseguenza, le funzioni statiche non hanno puntatore this, e qualsiasi accesso esplicito o implicito ad esso genera un errore in compilazione.

```
int queue::scrivi() {
  //cout << sloc;              // errore
  //cout << this->sloc;        // errore
  //cout << qget();            // errore
  //cout << this->qget()       // errore
  return count;                // OK
}
```

Le static member functions sono di particolare utilità nell'implementazione di alcuni design pattern (es. Singleton).

Friend functions

In C++ è possibile dichiarare una funzione non membro di una classe come **friend** di quella classe. La funzione dichiarata friend può accedere ai dati privati della classe.
La dichiarazione di una funzione friend viene effettuata inserendone il prototipo nella sezione public della dichiarazione di classe.

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

class myclass {
  int a, b;
public:
  myclass(int i, int j) { a=i; b=j; }
  friend int sum(myclass x);      // sum è una funzione friend di myclass
};


int sum(myclass x)
{
  return x.a + x.b;
}
```

```
int main()
{
  myclass mc(3, 4);

  cout << sum(mc);          // visualizza 7

  return 0;
}
```

Notare che:

- sum non è member function di nessuna classe, quindi viene invocata normalmente come una qualsiasi altra funzione, e non mediante un oggetto e l'operatore punto. Infatti, invocarla in questo modo causerebbe un errore di compilazione:

```
cout << mc.sum();    // errore!
```

- sum non è membro della classe myclass, ma per effetto della dichiarazione friend può accedere ai suoi dati privati.

Attraverso le friend functions è possibile aggirare le restrizioni di accesso ai dati privati delle classi. Questo concetto è in contrasto con la information hiding, concetto portante della programmazione ad oggetti.
Possiamo dire che il C++ non rappresenta un'implementazione pura della programmazione ad oggetti: si tratta piuttosto di un'implementazione orientata all'efficienza.

Is C++ an Object-Oriented language?
C++ is a multi-paradigm programming language that supports Object-Oriented and other useful styles of programming. If what you are looking for is something that forces you to do things in exactly one way, C++ isn't it. There is no one right way to write every program - and even if there were there would be no way of forcing programmers to use it.
The more general aim was to design a language in which I could write programs that were both efficient and elegant. Many languages force you to choose between those two alternatives.
(Bjarne Stroustrup)

Membri const e mutable

Oltre ad attributi di tipo static, è possibile avere attributi **const**.
In questo caso però l'attributo const non è trattato come una normale costante: esso viene allocato per ogni istanza come un normale attributo, tuttavia il valore che esso assume per ogni istanza viene stabilito una volta per tutte all'atto della creazione dell'istanza stessa, e non potrà mai cambiare durante la vita dell'oggetto.
Il valore di un attributo const va settato tramite la lista di inizializzazione del costruttore:

```
class MyClass {
private:
  const int ConstMember;          // attributo const
  float AFloat;

public:
  MyClass(int a, float b);
};

MyClass::MyClass(int a, float b)
     : ConstMember(a), AFloat(b) { };
```

Il motivo per cui bisogna ricorrere alla lista di inizializzazione è semplice: l'assegnamento è una operazione proibita sulle costanti, sulle quali è invece obbligatoria l'inizializzazione.

E' anche possibile avere *metodi const*, analogamente a quanto avviene per le static member functions.
Dichiarando un metodo const si stabilisce un contratto con il compilatore: la funzione membro si impegna a non accedere in scrittura ad un qualsiasi dato della classe, e il compilatore si impegna a segnalare con un errore ogni tentativo in tal senso. Oltre a ciò esiste un altro vantaggio a favore dei metodi const: sono gli unici a poter essere eseguiti su istanze costanti (che per loro natura non possono essere modificate).
Per dichiarare una funzione membro const è necessario far seguire la lista dei parametri dalla keyword const, come mostrato nel seguente esempio:

Infine, è anche possibile creare *oggetti const*, ossia istanze costanti di una classe, sulle quali è possibile invocare esclusivamente metodi const.

```
class MyClass {
private:
    const int ConstMember;          // attributo const
    float AFloat;

public:
    MyClass(int a, float b);
    int GetConstMember() const;     // dichiarazione metodo const
    void ChangeFloat(float b);
};


MyClass::MyClass(int a, float b)
     : ConstMember(a), AFloat(b) { };


int MyClass::GetConstMember() const {
    return ConstMember;             // implementazione metodo const
}

void MyClass::ChangeFloat(float b); {
     AFloat = b;
}
```

```
int main() {
    MyClass A(1, 5.3);
    const MyClass B(2, 3.2);        // oggetto const

    A.GetConstMember();         // OK
    B.GetConstMember();         // è un metodo const → OK
    A.ChangeFloat(1.2);         // OK
    //B.ChangeFloat(1.7);       // Errore!
    return 0;
}
```

Come per i metodi static, non è possibile avere costruttori e distruttori const (sebbene essi vengano utilizzati per costruire e distruggere anche le istanze costanti).

Talvolta può essere necessario che un metodo const possa modificare uno o più attributi della classe. Situazioni di questo genere sono rare ma possibili.

Per rendere esplicita una situazione di questo tipo è stata introdotta la keyword **mutable**. Un attributo dichiarato mutable può essere modificato anche da metodi const.

Ad esempio, consideriamo il caso di una classe che debba tenere traccia del numero di accessi in lettura ai suoi dati.
I dati della classe sono costanti, ad eccezione del contatore del numero di accessi. L'utilizzo dello specificatore mutable ci permette di realizzare una soluzione compatta ed elegante.

```
class AccessCounter {
private:
    const double PI;
    mutable int Counter;            // attributo modificabile anche
                                    // da metodi const
public:
    AccessCounter();
    double GetPIValue() const;
    int GetAccessCount() const;
};


AccessCounter::AccessCounter() : PI(3.14159265),
                                 Counter(0) {}

double AccessCounter::GetPIValue() const {
    ++Counter;        // OK
    return PI;
}

int AccessCounter::GetAccessCount() const {
    return Counter;
}
```