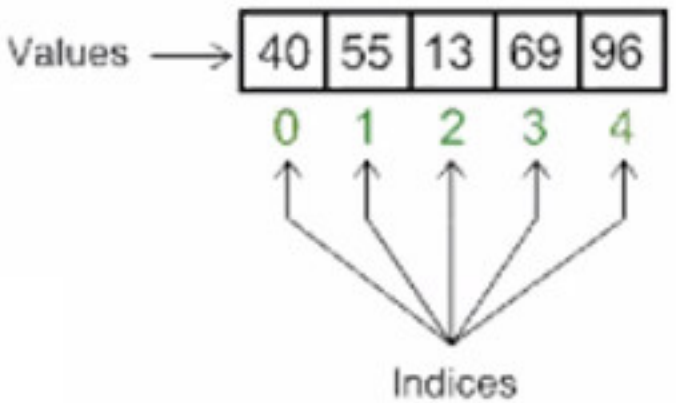
Definire una classe **Vettore** i cui oggetti rappresentano array di interi. Vettore deve includere un costruttore di default, una operazione di concatenazione che restituisce un nuovo vettore v1+v2, una operazione di append v1.append(v2), l'overloading dell'uguaglianza, dell'operatore di output e dell'operatore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



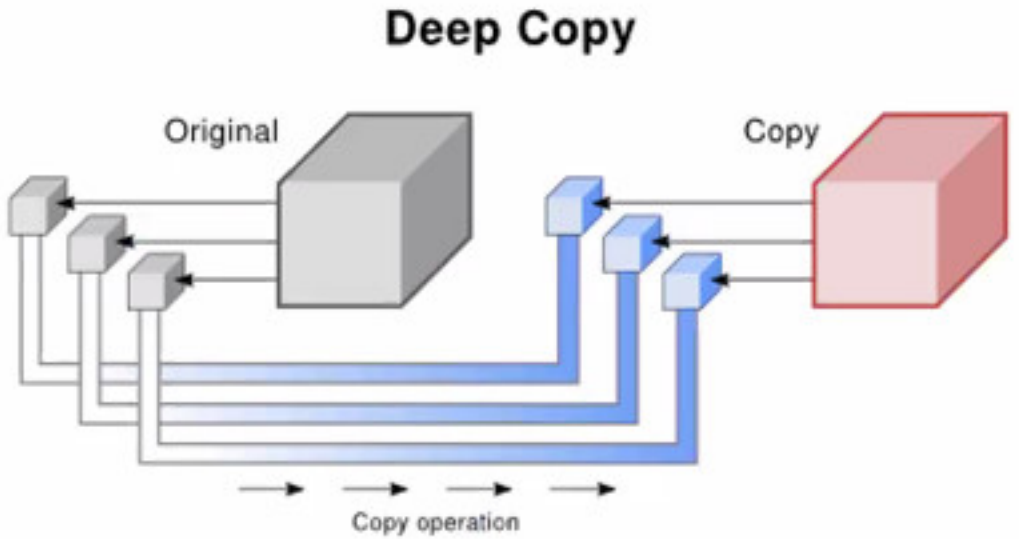
```
/* ESERCIZIO:
Definire una classe vettore i cui oggetti rappresentano array di interi.
vettore deve includere un costruttore di default, una operazione di
concatenazione che restituisce un nuovo vettore v1+v2, una operazione di
append v1.append(v2), l'overloading dell'uguaglianza, dell'operatore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

#include<iostream>

class Vettore {
private:
  int* a;
  unsigned int size; // size ≧ 0  (garantito da unsigned int)
  // vettore vuoto IFF a==nullptr && size == 0
  // vettore non vuoto IFF a!=nullptr && size>0
public:
  // unsigned int => Vettore
  Vettore(unsigned int s =0, int init=0): a(s==0 ?  nullptr : new int[s]), size(s) {
    for(int j=0; j<size; ++j) a[j]=init;
  }

  Vettore(const Vettore& v): a(v.size == 0 ? nullptr : new int[v.size]), size(v.size) {
    for(unsigned int j=0; j<size; ++j) a[j]=v.a[j];
  }

  Vettore& operator=(const Vettore& v) {
    if (this != &v) {
      delete[] a; // attenzione: delete[] e NON delete
      size = v.size;
      a = size == 0 ? nullptr : new int[size];
      for (int i = 0; i < size; i++) a[i] = v.a[i];
    }
    return *this;
  }

  ~Vettore() {if(a) delete[] a;}

  Vettore& append(const Vettore& v) {
    if (v.size != 0){
      int* aux = new int[size+v.size];
      for (int i = 0; i < size; i++) aux[i] = a[i];
      for (int i = 0; i < v.size; i++) aux[size+i] = v.a[i];
      size += v.size;
      delete[] a; // FONDAMENTALE
      a = aux;
    }
    return *this;
  }

  Vettore operator+(const Vettore& v) const {
    Vettore aux(*this);
    aux.append(v);
    return aux;
  }

  bool operator==(const Vettore& v) const {
    if(this == &v) return true;
    if(size!=v.size) return false;
    // size == v.size >= 0
    for(int i=0;i<size;i++)
      if(a[i]!=v.a[i]) return false;
    // forall i in [0,size-1], a[i]==v.a[i]
    return true;
  }

  int& operator[](unsigned int i) const {
    return *(a+i);
  }

  unsigned int getSize() const {
    return size;
  } //FINE CLASSE
```

```
};


std::ostream& operator<<(std::ostream& os, const Vettore& v) {
  os << '[';
  int i = 0;
  while(i<(v.getSize())) {
    os << v[i] << ((i!=v.getSize()-1)  ? ',' : ']' );
    i++;
  }
  if(v.getSize()==0) os << ']';
  return os;
}

int main() {
  Vettore v1(4), v2(3,2), v3(5,-3);
  v1 = v2+v3;
  v2.append(v2);
  v3.append(v1).append(v3);
  std::cout << v1 << std::endl;
  std::cout << v2 << std::endl;
  std::cout << v3 << std::endl;
}
```

```
class S {
public:
  string s;
  S(string t): s(t) {}
};
class N {
private:
  S x;
public:
  N* next;
  N(S t, N* p): x(t), next(p) {cout << "N2 ";}
  ~N() {if (next) delete next; cout << x.s + "~N ";}
};
class C {
  N* pointer;
public:
  C(): pointer(0) {}
  ~C() {delete pointer; cout << "~C ";}
  void F(string t1, string t2 = "pippo") {
    pointer = new N(S(t1),pointer);
    pointer = new N(t2,pointer);
  }
};
main(){
  C* p = new C; cout << "UNO\n";
  p->F("pluto","paperino"); p->F("topolino"); cout <<"DUE\n";
  delete p; cout <<"TRE\n";
}
```


```
NESSUNA STAMPA UNO
N2 N2 N2 N2 DUE
pluto~N paperino~N topolino~N pippo~N ~C TRE
NESSUNA STAMPA
```

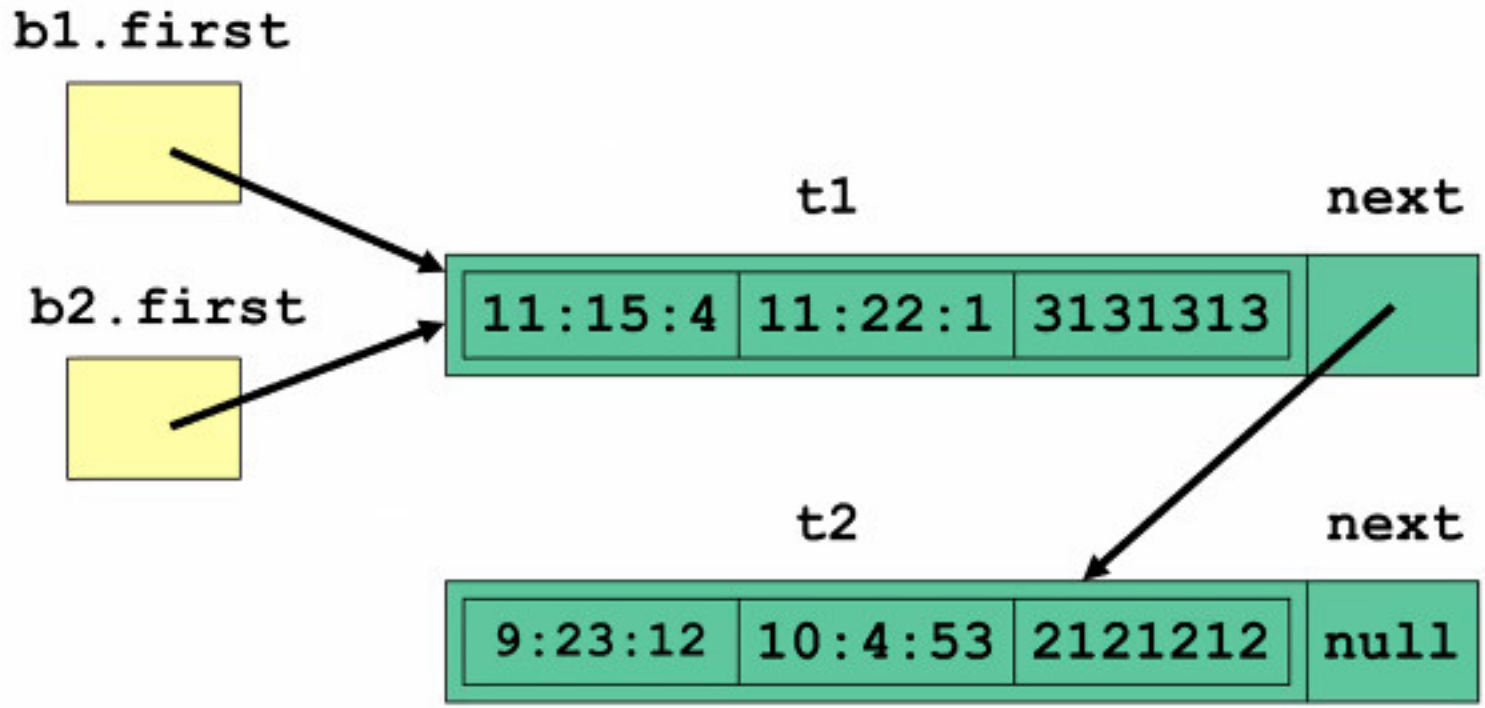
Le copie profonde sono una soluzione ottimale?



Soluzione costosa, talvolta inaccettabile,

a volte inutile

Situazione di condivisione di memoria

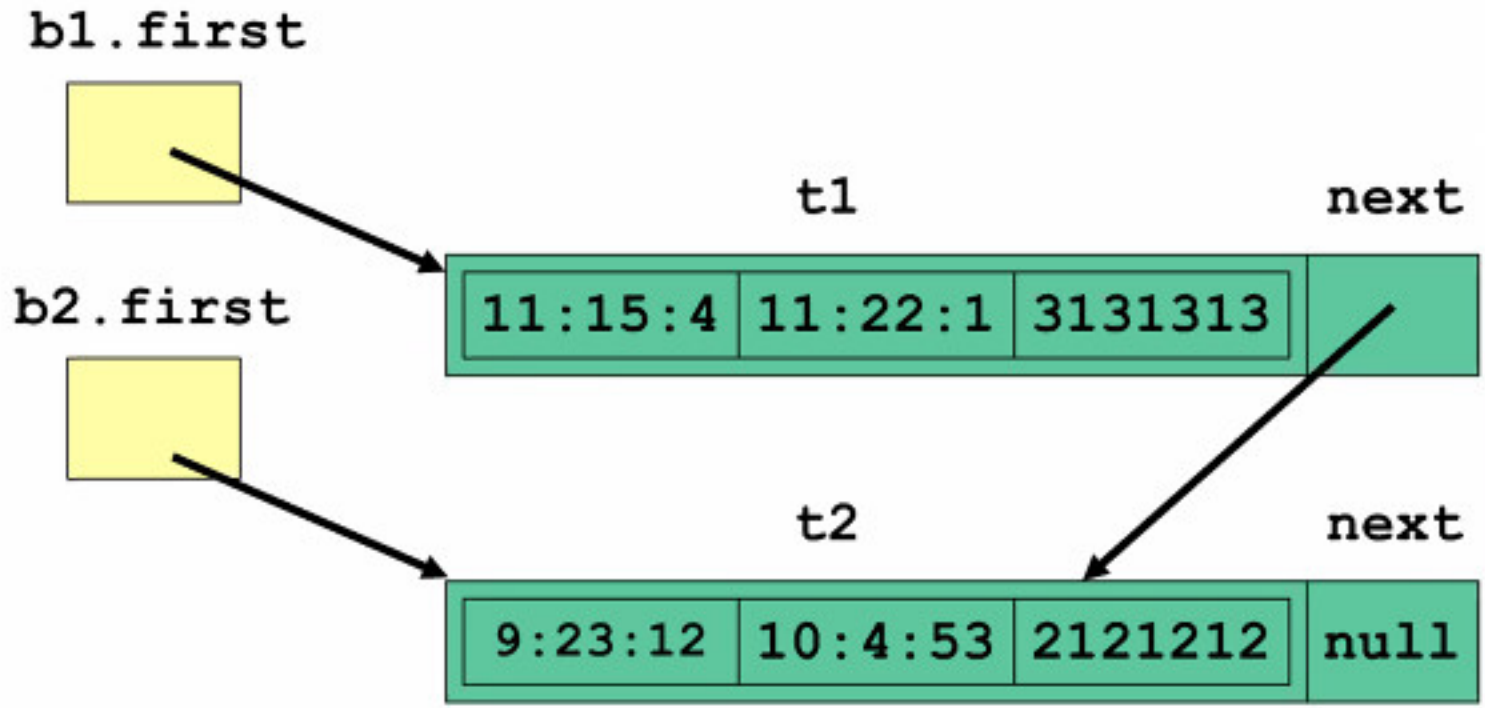


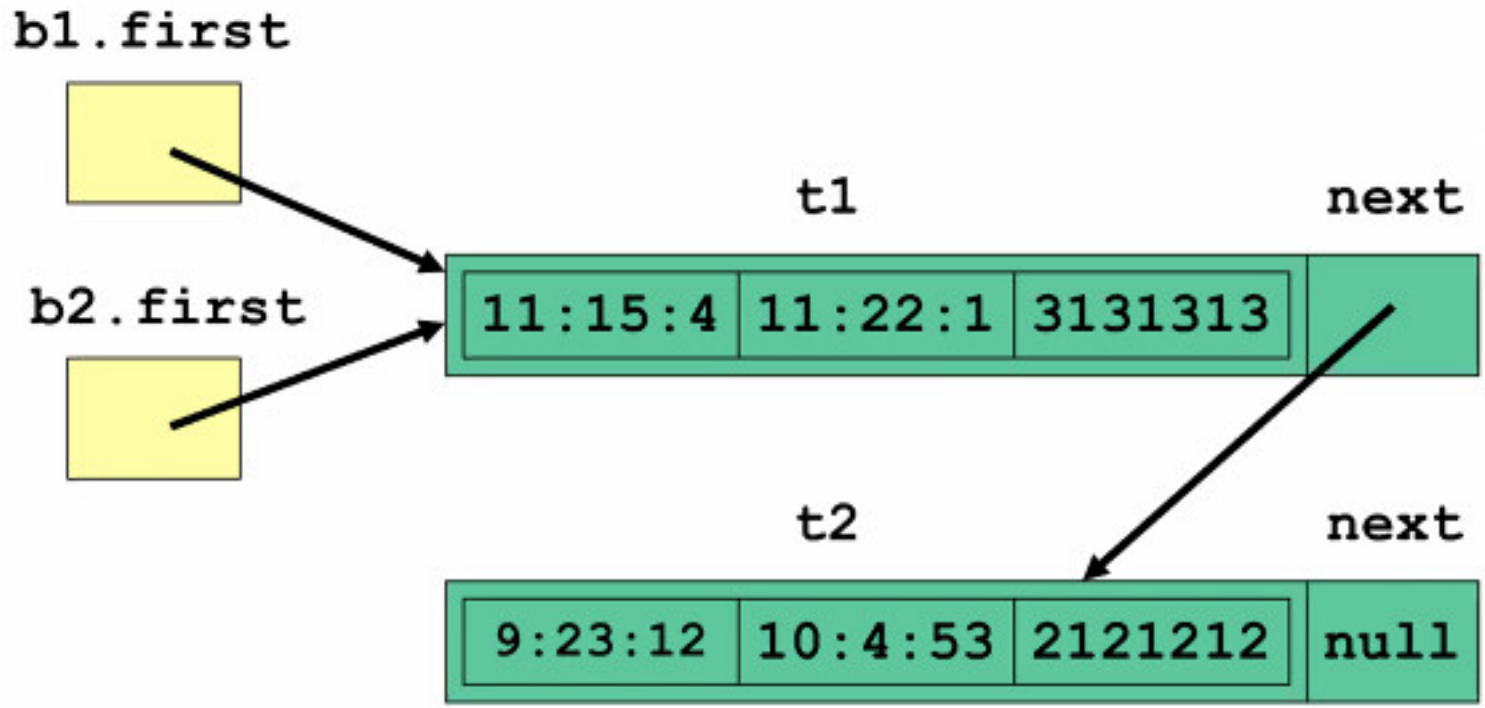
Cosa si può fare per mantenere uno stato di condivisione della memoria consistente dopo

`b2.Togli_Telefonata(t1);` ?

Vorremmo la seguente situazione dopo

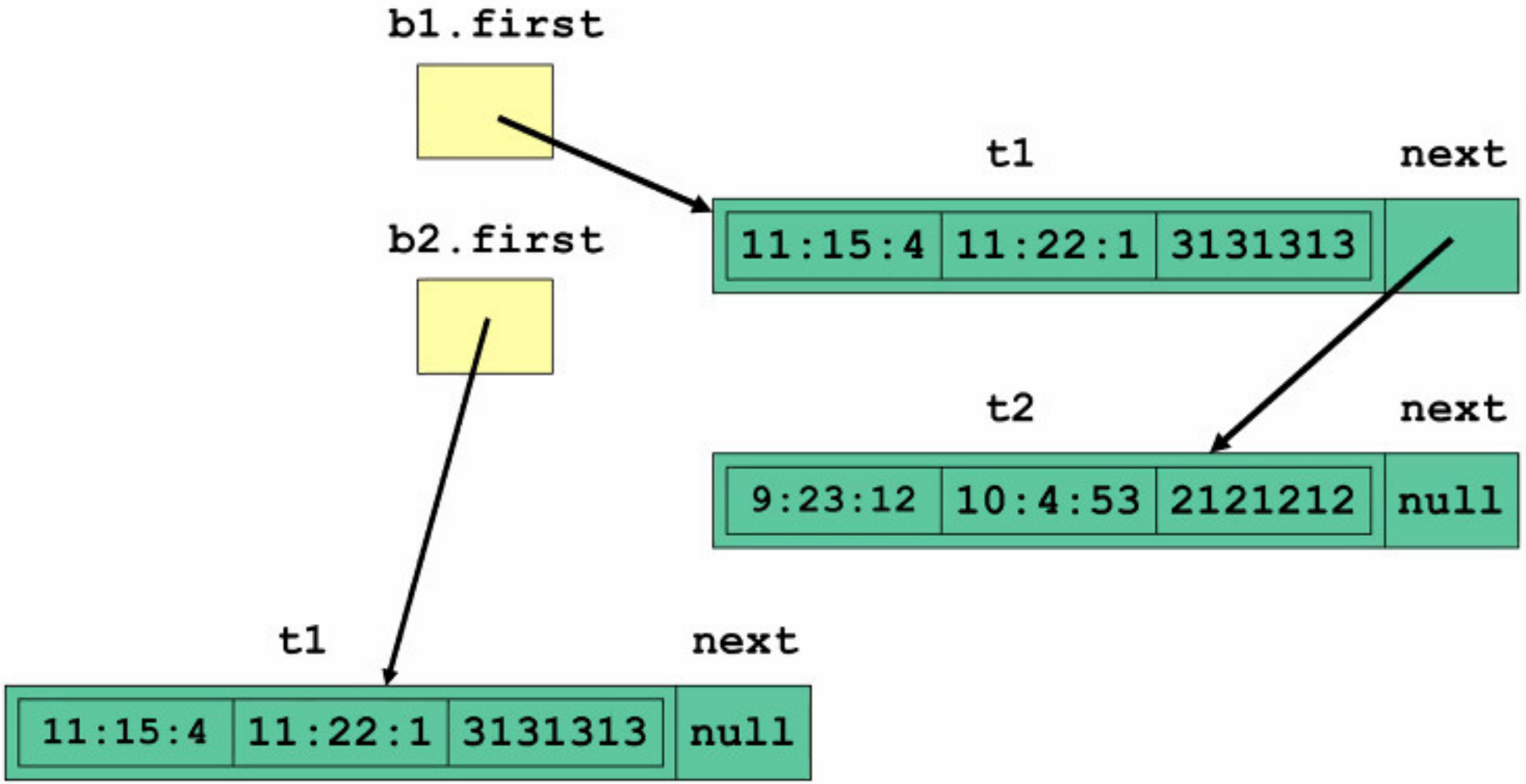
```
b2.Togli_Telefonata(t1);
```





Invece cosa si può fare dopo `b2.Togli_Telefonata(t2);` ?

Dopo `b2.Togli_Telefonata(t2);` vogliamo la seguente situazione:



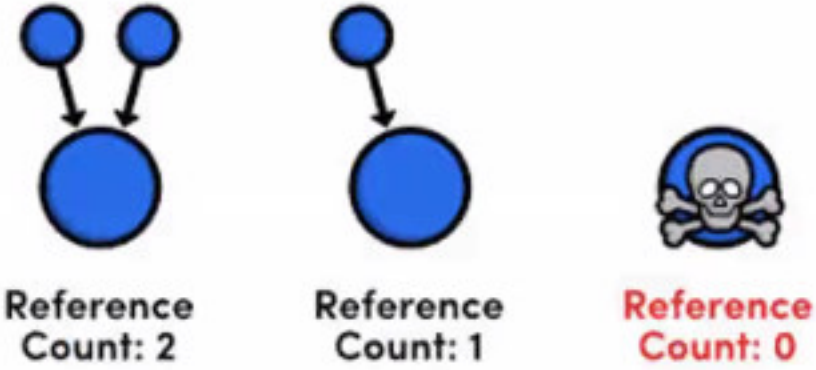
Reference counting

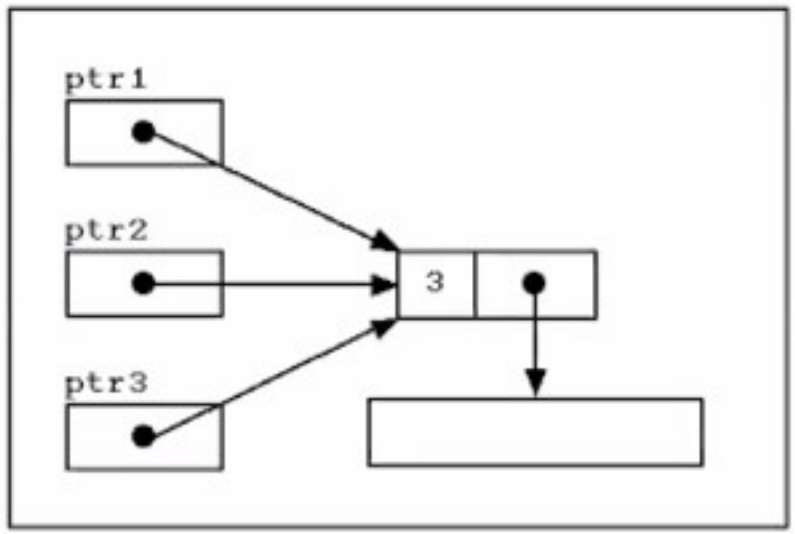
From Wikipedia, the free encyclopedia

This article **needs additional citations for verification**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed. *(November 2009)*

In computer science, **reference counting** is a technique of storing the number of references, pointers, or handles to a resource such as an object, block of memory, disk space or other resource. It may also refer, more specifically, to a garbage collection algorithm that uses these reference counts to deallocate objects which are no longer referenced.

Contents [hide]


Reference
Count: 2
Reference
Count: 1
Reference
Count: 0


ptr1
ptr2
ptr3
3

Si incapsula in una classe il puntatore **`nodo*`** e si ridefinisce assegnazione, costruttore di copia e distruttore. Si definisce un cosiddetto "**smart pointer**". Gli smart pointer dovranno essere dotati di una interfaccia pubblica che permetta all'utente di utilizzarli come fossero dei puntatori ordinari.

Smart pointer

From Wikipedia, the free encyclopedia

In computer science, a **smart pointer** is an abstract data type that simulates a pointer while providing added features, such as automatic memory management or bounds checking. Such features are intended to reduce bugs caused by the misuse of pointers, while retaining efficiency. Smart pointers typically keep track of the memory they point to, and may also be used to manage other resources, such as network connections and file handles. Smart pointers originated in the programming language C++.

L'idea degli smart pointer è stata implementata in **C++11**

Dynamic memory management

Smart pointers

Smart pointers enable automatic, exception-safe, object lifetime management.

Defined in header <memory>

Pointer categories

`unique_ptr` (C++11)	smart pointer with unique object ownership semantics (class template)
`shared_ptr` (C++11)	smart pointer with shared object ownership semantics (class template)
`weak_ptr` (C++11)	weak reference to an object managed by `std::shared_ptr` (class template)
`auto_ptr` (removed in C++17)	smart pointer with strict object ownership semantics (class template)

std::shared_ptr

Defined in header <memory>

```
template< class T > class shared_ptr;    (since C++11)
```

`std::shared_ptr` is a smart pointer that retains shared ownership of an object through a pointer. Several `shared_ptr` objects may own the same object. The object is destroyed and its memory deallocated when either of the following happens:

- the last remaining `shared_ptr` owning the object is destroyed;
- the last remaining `shared_ptr` owning the object is assigned another pointer via `operator=` or `reset()`.

L'idea degli smart pointer è stata implementata in **C++11**

C++11 rende disponibili i tipi `std::shared_ptr<T>` e `std::weak_ptr<T>` che implementano dei puntatori smart ad un generico tipo `T` con reference counting basandosi sull'implementazione già disponibile nella libreria Boost.

Boost (C++ libraries)

From Wikipedia, the free encyclopedia
(Redirected from Boost library)

Boost is a set of libraries for the C++ programming language that provide support for tasks and structures such as linear algebra, pseudorandom number generation, multithreading, image processing, regular expressions, and unit testing. It contains over eighty individual libraries.

Most of the Boost libraries are licensed under the Boost Software License, designed to allow Boost to be used with both free and proprietary software projects. Many of Boost's founders are on the C++ standards committee, and several Boost libraries have been accepted for incorporation into both Technical Report 1 and the C++11 standard.[1]

Boost

Boost logo

Stable release 1.55.0 / November 11, 2013; 3 months ago