Definire un template di classe `Vettore<T,sz>` i cui oggetti rappresentano un array di tipo `T` e dimensione `sz` ≥ 0. Il template di classe `Vettore<T,sz>` deve includere: un costruttore `Vettore(const T& x)` che costruisce un array in cui tutte le celle memorizzano il valore `x`; un costruttore di default `Vettore()` che costruisce un array in cui tutte le celle memorizzano il valore di default `T()` del tipo `T`; ridefinizione di costruttore di copia profonda, assegnazione profonda e distruzione profonda; overloading degli operatori `operator*` di dereferenziazione e `operator[]` di indicizzazione con comportamento analogo ai corrispondenti operatori disponibili per gli array ordinari; overloading dell'operatore di output.

Ad esempio, il seguente codice dovrà compilare correttamente e l'esecuzione dovrà **provocare esattamente** le stampe riportate nei commenti.

```
Vettore<int,4> v1(2); Vettore< Vettore<int,3> ,3> v2(3); Vettore<int,4> v3(v1); Vettore<int,4> v4;
v3[2]=6;
*v1=9;
v4=v1;
v4[3]=5
std::cout << v1 << std::endl; // 9 2 2 2
std::cout << v2 << std::endl; // 3 3 3  3 3 3  3 3 3
std::cout << v3 << std::endl; // 2 2 6 2
std::cout << v4 << std::endl; // 9 2 2 5
```

STANDARD

COPIA

ASSEGNAZIONE (COPIA) $v_4 = v_1 \equiv v_4(v_1)$

<<, [], * → 3 OPERATORI

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

template <class T = int, unsigned int size = 0>
class Vettore {
        private:
                T* ptr; // andava bene anche int* ptr;
        public:
                // Costruttore di default ridefinito
                Vettore(const T& x = T()): ptr(size == 0) ? nullptr : new T[size]){
                        for(int i = 0; i < size; i++){
                                ptr[i] = x;
                        }
                }
                // Esempio di utilizzo → Vettore<int,4> v1(2);


                // Costruttore di copia
                Vettore(const Vettore& x): ptr(size == 0) ? nullptr : new T[size]){
                        for(int i = 0; i < size; i++){
                                ptr[i] = x.ptr[i];
                }

                // Assegnazione standard
                Vettore& operator=(const Vettore& x){
                        if(this ≠ &x){          // Controllo importante
                                // (1) Cancellazione - ptr
                                delete[] ptr;
                                // erase: si usa SOLO per vector / list / queue ...

                                // (2) Costruirlo interamente

                                // Lo devo creare da zero ...  Notazione compatta
                                ptr = size == 0 ? nullptr : new T[size];

                                // Alternativamente
                                if(size == 0) ptr = nullptr;
                                else T* ptr = new T[size];

                                // Assegnazione di ogni valore di x
                                for(int i = 0; i < size; i++)
                                        ptr[i] = x.ptr[i];
                        }
                        (3) Ritorno
                        return *this;
                }
};
```

NON NEGATIVA

R VALORE COSTANTE TEMPLATE

*v1=9; → [ ], *

v1[0]

```
T& operator[](unsigned int i) {
  return a[i];
}

T& operator*() {
  return a[0];
}
```

v[_ _ [i] _] = SUBSCRIPT

↑ BEGIN = v[0]

```
const T& operator[](unsigned int i) const {
  return a[i];
}
T& operator[](unsigned int i) {
  return a[i];
}
```

VOLENDO, È CONST

v4[3] = 5;

↑ SUBSCRIPTING

```
std::cout << v1 << std::endl; // 9 2 2 2
std::cout << v2 <<std::endl;  // 3 3 3  3 3 3  3 3 3
std::cout << v3 << std::endl; // 2 2 6 2
std::cout << v4 << std::endl; // 9 2 2 5
```

STAMPA → SO PARTO DA TEMPLATE

VA DEFINITA FUORI DALLA CLASSE

CLASS VETTORE{.

...

};

// FUORI → OPERATORE DI STAMPA

STD::COUT << V1 << STD::ENDL;

```
template <class T, unsigned int sz>
std::ostream& operator<<(std::ostream& os, const Vettore<T,sz>& v) {
  for(int i=0; i<sz; ++i) os << v[i] << ' ';
  return os;
}
```

PARTO DI TEMPLATE = FUORI CLASSE