```
/*
Definire un template di classe albero<T> i cui oggetti rappresentano
un albero 3-ario ove i nodi memorizzano dei valori di tipo T ed hanno
3 figli (invece dei 2 figli di un usuale albero binario). Il template
albero<T> deve soddisfare i seguenti vincoli:
1. Deve essere disponibile un costruttore di default che costruisce l'albero vuoto.
2. Gestione della memoria senza condivisione.
3. Metodo void insert(const T&): a.insert(t) inserisce nell'albero a una nuova radice che memorizza il
valore t ed avente come figli 3 copie di a
4. Metodo bool search(const T&): a.search(t) ritorna true se t occorre nell'albero a, altrimenti ritorna
false.
5. Overloading dell'operatore di uguaglianza.
6. Overloading dell'operatore di output.
*/

#include<iostream>

template <class T> class albero; // dichiarazione incompleta

template<class T>
std::ostream& operator<<(std::ostream& os, const albero<T>& a);

template <class T>
class albero {
  friend std::ostream& operator<< <T> (std::ostream&, const albero&);
  private:
  // classe annidata associata
  class nodo {
  public:
    T info;
    nodo *sx, *cx, *dx;
    nodo(const T& x, nodo* s =0, nodo* c =0, nodo* d =0):
      info(x), sx(s), cx(c), dx(d) {}
  };

  nodo* root;

  // copia profonda ricorsiva
  static nodo* copia(nodo* r) {
    if(!r) return nullptr;
    // albero non vuoto
    return new nodo(r->info, copia(r->sx), copia(r->cx), copia(r->dx));
  }

  // distruzione profonda ricorsiva
  static void distruggi(nodo* r) {
    if(r) {
      distruggi(r->sx); distruggi(r->cx); distruggi(r->dx);
      delete r;
    }
  }

  static bool search_rec(nodo* r, const T& t) {
    if(!r) return false;
    // r punta alla radice di un albero non vuoto
    return r->info == t || search_rec(r->sx,t) || search_rec(r->cx,t) || search_rec(r->dx,t);
  }

  static bool equal_rec(nodo* r1, nodo* r2) {
    if(!r1 && !r2) return true;
    // r1 | r2
    if(!r1 || !r2) return false;
    // r1 & r2, T deve supportare operator==
    return r1->info == r2->info && equal_rec(r1->sx,r2->sx) &&
      equal_rec(r1->cx,r2->cx) && equal_rec(r1->dx,r2->dx);
  }

  static std::ostream& print_rec(std::ostream& os, nodo* r){
    // caso base: albero vuoto
    if(!r) return os;
    // passo induttivo: albero non vuoto
    os << r->info << " ";  // T deve supportare operator<<
    print_rec(os,r->sx);
    print_rec(os,r->cx);
    return print_rec(os,r->dx);
  }
```

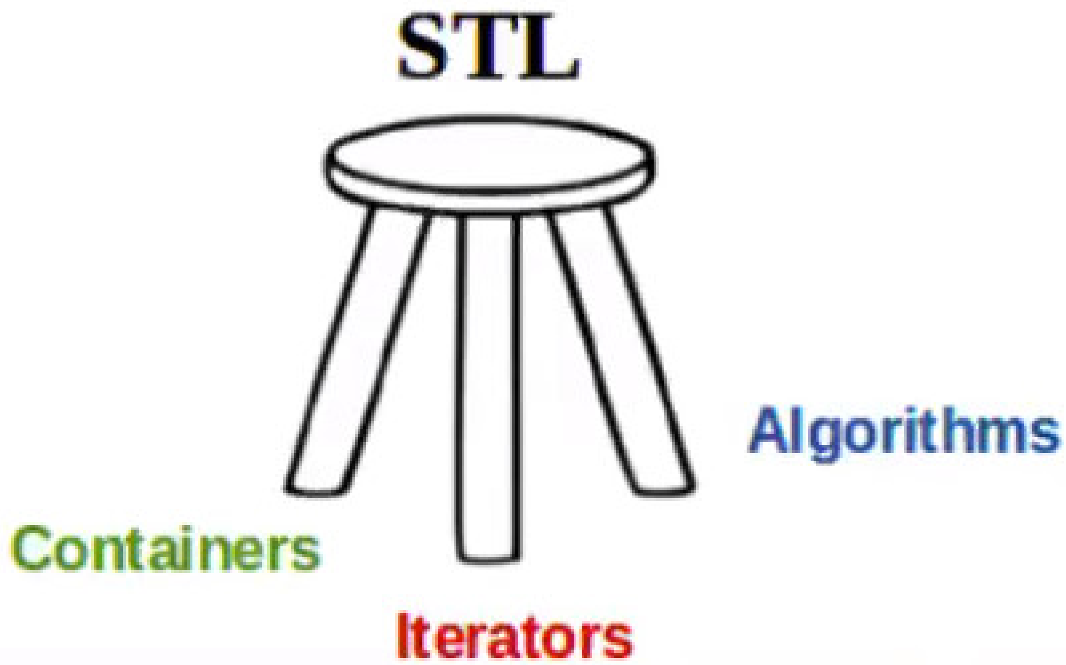
```
public:
  albero(): root(nullptr) {}

  albero(const albero& a): root(copia(a.root)) {}

  albero& operator=(const albero& a) {
    if(this != &a) {
      if(root) distruggi(root);
      root = copia(a.root);
    }
    return *this;
  }

  ~albero() {if(root) distruggi(root);}

  void insert(const T& x) {
    root = new nodo(x,copia(root), copia(root), root);
  }

  bool search(const T& t) const {
    return search_rec(root,t);
  }

  bool operator==(const albero& a) const {
    return equal_rec(root,a.root);
  }
};

template<class T>
std::ostream& operator<<(std::ostream& os, const albero<T>& a) {
  return albero<T>::print_rec(os,a.root);
}

int main() {
  albero<char> t1, t2, t3;
  t1.insert('b');
  t1.insert('a');
  t2.insert('a');
  t3 = t1;
  t3.insert('c');
  std::cout << (t1 == t2) << std::endl;
  std::cout << t1.search('b') << std::endl;
  std::cout << t1 << std::endl << t2 << std::endl << t3 << std::endl;
}
```

```
/*
Si considerino le seguenti definizioni. Fornire una dichiarazione
(non e` richiesta la definizione) dei membri pubblici della classe Z
nel minor numero possibile in modo tale che la compilazione del
main() non produca errori. Attenzione: ogni dichiarazione in Z
non necessaria per la corretta compilazione del main() e'
penalizzata.
*/

class Z {
public:
  int& operator++();
  int operator++(int);
  bool operator==(const Z&) const;
  Z(const int&); // agisce da convertitore int => Z
};

template <class T1, class T2 =Z>
class C {
public:
  T1 x;
  T2* p;
};

template<class T1,class T2>
void fun(C<T1,T2>* q) {
  ++(q->p); // nessun requirement
  if(true == false) cout << ++(q->x); // q->x di tipo T1, operator++() T1
  else cout << q->p; // nessun requirement
  (q->x)++; // operator++(int) su T1
  if(*(q->p) == q->x) *(q->p) = q->x; // (1) bool operator==(T2,T1), (2) operator=(T2,const T1&)
  T1* ptr = &(q->x); // nessun requirement
  T2 t2 = q->x; // T2(const T1&)
}

main(){
  C<Z> c1; fun(&c1); C<int> c2; fun(&c2);
  // C<Z,Z> c1;
  // fun<Z,Z>, i.e. T1=Z, T2=Z
  // C<int,Z> c2;
  // fun<int,Z>, i.e. T1=int, T2=Z
}
```


IT'S YOUR
LIBRARY
USE IT!

Standard Template Library

STL



C++ Standard Library

From Wikipedia, the free encyclopedia

In the C++ programming language, the **C++ Standard Library** is a collection of classes and functions, which are written in the core language and part of the C++ ISO Standard itself.[1] The C++ Standard Library provides several generic containers, functions to utilize and manipulate these containers, function objects, generic strings and streams (including interactive and file I/O), support for some language features, and everyday functions for tasks such as finding the square root of a number. The C++ Standard Library also incorporates 18 headers of the ISO C90 C standard library ending with ".h", but their use is deprecated.[2] No other headers in the C++ Standard Library end in ".h". Features of the C++ Standard Library are declared within the `std` namespace.

The C++ Standard Library is based upon conventions introduced by the Standard Template Library (STL), and has been influenced by research in generic programming and developers of the STL such as Alexander Stepanov and Meng Lee.[3][4] Although the C++ Standard Library and the STL share many features, neither is a strict superset of the other.

A noteworthy feature of the C++ Standard Library is that it not only specifies the syntax and semantics of generic algorithms, but also places requirements on their performance.[5] These performance requirements often correspond to a well-known algorithm, which is expected but not required to be used. In most cases this requires linear time $O(n)$ or linearithmic time $O(n \log n)$, but in some cases higher bounds are allowed, such as quasilinear time $O(n \log^2 n)$ for stable sort (to allow in-place merge sort). Previously sorting was only required to take $O(n \log n)$ on average, allowing the use of quicksort, which is fast in practice but has poor worst-case performance, but introsort was introduced to allow both fast average performance and optimal worst-case complexity, and as of C++11, sorting is guaranteed to be at worst linearithmic. In other cases requirements remain laxer, such as selection, which is only required to be linear on average (as in quicksort),[6] not requiring worst-case linear as in introselect.

The C++ Standard Library underwent ISO standardization as part of the C++ ISO Standardization effort, and is undergoing further work[7] regarding standardization of expanded functionality.