```cpp
#include <iostream>
#include <cassert>

using namespace std;

template<class T>
class Node{
public:
  Node(T aKey=T()):mKey(aKey),mNext(0),mPrev(0){}
public:
  T mKey;
  Node* mNext;
  Node* mPrev;
};

template<class T> class List;
template<class T> ostream& operator<<(ostream& out, const List<T>& aList);

template<class T>
class List{
  friend ostream& operator<< <T> (ostream& out, const List<T>& aList);
  friend istream& operator>>(istream& in, List<T>& aList)
  {
    int key;
    while (in>>key) aList.Insert(key);
    return in;
  }

public:
  List();
  ~List();
  List(const List& aList);
  List& operator=(const List& aList);
  void Insert(T aKey);
  void Delete(int aIndex);
  void Output(ostream& out) const;
  T& operator[](int aIndex);
  bool IsEmpty();
private:
  void Init();
  Node<T>* Find(int aIndex);
private:
  Node<T>* mHead;
};

template<class T>
bool List<T>::IsEmpty()
{
  return mHead->mNext==mHead;
}

template<class T>
void List<T>::Init()
{
  mHead=new Node<T>(T());
  mHead->mNext=mHead;
  mHead->mPrev=mHead;
```

```
}

template<class T>
List<T>::List()
{
  Init();
}

template<class T>
List<T>::~List()
{
  Node<T>* cursor=mHead;
  do
    {
      Node<T>* tmp=cursor->mNext;
      delete cursor;
      cursor=tmp;
    } while (cursor!=mHead);
}

template<class T>
List<T>::List(const List& aList)
{
  Init();
  *this=aList;
}

template<class T>
List<T>& List<T>::operator=(const List& aList)
{
  if (!IsEmpty()) delete this;
  Init();
  Node<T>* cursor=aList.mHead->mNext;
  do
    {
      Insert(cursor->mKey);
      cursor=cursor->mNext;
    } while (cursor!=aList.mHead);
  return *this;
}

template<class T>
void List<T>::Insert(T aKey)
{
  Node<T>* new_node=new Node<T>(aKey);
  mHead->mPrev->mNext=new_node;
  new_node->mNext=mHead;
  new_node->mPrev=mHead->mPrev;
  mHead->mPrev=new_node;
}

template<class T>
void List<T>::Delete(int aIndex)
{
  assert(aIndex>=0);
  Node<T>* cursor=Find(aIndex-1);
  assert(cursor!=0);
```

```
  Node<T>* tmp=cursor->mNext;
  cursor->mNext=cursor->mNext->mNext;
  tmp->mNext->mPrev=cursor;
  delete tmp;
}

template<class T>
T& List<T>::operator[](int aIndex)
{
  assert(aIndex>=0);
  Node<T>* cursor=Find(aIndex);
  assert(cursor!=0);
  return cursor->mKey;
}

template<class T>
Node<T>* List<T>::Find(int aIndex)
{
  int index=-1;
  Node<T>* cursor=mHead;
  do
    {
      if (index==aIndex) return cursor;
      index++;
      cursor=cursor->mNext;
    } while (cursor!=mHead);
  return 0;
}

template<class T>
void List<T>::Output(ostream& out)const
{
  Node<T>* cursor=mHead->mNext;
  do
    {
      out<<cursor->mKey<<" ";
      cursor=cursor->mNext;
    } while (cursor!=mHead);
}

template<class T>
ostream& operator<<(ostream& out, const List<T>& aList)
{
  aList.Output(out);
  return out;
}

int main(){
  List<char> list;
  for (int i=0;i<10;i++) list.Insert('a'+i);
  cout<<list<<endl;
  cout<<"Elemento di posizione 4: "<<list[4]<<endl;
  cout<<"Elemento di posizione 9: "<<list[9]<<endl;
  cout<<"Cancella elemento di posizione 4"<<endl;
  list.Delete(4);
  cout<<"Elemento di posizione 4: "<<list[4]<<endl;
  cout<<list<<endl;
```

```
  cout<<"Cancella elemento di posizione 8"<<endl;
  list.Delete(8);
  cout<<list<<endl;
  cout<<"Cancella elemento di posizione 0"<<endl;
  list.Delete(0);
  cout<<list<<endl;
  cout<<"Copia lista"<<endl;
  List<char> list2;
  List<char> list3(list);
  list2=list;
  cout<<list2<<endl;
  cout<<list3<<endl;
  return 0;
}
```