```
//=================================
// SListInt.h
//=================================
// Declarations for SListInt

#ifndef SLIST_INT_H
#define SLIST_INT_H

class SListInt {
  //-----------------------------------------------------
  // Constructor and destructor
  //-----------------------------------------------------

 public:
  SListInt();
  ~SListInt();

  //-----------------------------------------------------
  // Copy constructor, swap, assignment operator
  //-----------------------------------------------------

 public:
  SListInt(const SListInt& lst);
  void      swap(SListInt& lst);
  SListInt& operator=(const SListInt& lst);

  //-----------------------------------------------------
  // Member functions
  //-----------------------------------------------------

 public:
  void push_front(int v);
  int  size() const;
  int  at(int idx) const;
  int& at(int idx);
  void reverse_v1();
  void reverse_v2();
  void print() const;

  //-----------------------------------------------------
  // Private member functions and fields
  //-----------------------------------------------------

 private:
  struct Node {
    int   value;
    Node* next_p;
  };

  Node* m_head_p;
};

#endif /* SLIST_INT_H */
```

```
//=================================
// SList.h
//=================================
// Declarations for SList<T>

#ifndef SLIST_H
#define SLIST_H
template <typename T>
class SList {
  //-----------------------------------------------------
  // Constructor and destructor
  //-----------------------------------------------------

 public:
  SList();
  ~SList();

  //-----------------------------------------------------
  // Copy constructor, swap, assignment operator
  //-----------------------------------------------------

 public:
  SList(const SList& lst);
  void   swap(SList& lst);
  SList& operator=(const SList& lst);

  //-----------------------------------------------------
  // Member functions
  //-----------------------------------------------------

 public:
  void     push_front(const T& v);
  int      size() const;
  const T& at(int idx) const;
  T&       at(int idx);
  void     reverse_v1();
  void     reverse_v2();
  void     print() const;

  //-----------------------------------------------------
  // Private member functions and fields
  //-----------------------------------------------------

 private:
  struct Node {
    T     value;
    Node* next_p;
  };

  Node* m_head_p;
};

  //-----------------------------------------------------
  // Include inline definitions
  //-----------------------------------------------------
#include "SList.inl"

#endif /* SLIST_H */
```

```
//========================================================================
// swap.inl
//========================================================================
// Implementation for generic swap function

template <typename T>
void swap(T& a, T& b) {
  //''' LAB TASK '''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // Implement generic swap
  //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // BEGIN REMOVE
  T tmp = a;
  a     = b;
  b     = tmp;
  // END REMOVE
}

//------------------------------------------------------------------------
// SList Destructor
//------------------------------------------------------------------------

template <typename T>
SList<T>::~SList() {
  //''' LAB TASK '''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // Implement destructor
  //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  while (m_head_p != nullptr) {
    // BEGIN REMOVE
    Node* temp_p = m_head_p->next_p;
    delete m_head_p;
    m_head_p = temp_p;
  }
  // END REMOVE
}

//------------------------------------------------------------------------
// SList<T>::push_front
//------------------------------------------------------------------------

template <typename T>
void SList<T>::push_front(const T& v) {
  //''' LAB TASK '''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // Implement push_front
  //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // BEGIN REMOVE
  Node* new_node_p   = new Node;
  new_node_p->value  = v;
  new_node_p->next_p = m_head_p;
  m_head_p           = new_node_p;
  // END REMOVE
}
```

```
//------------------------------------------------------------------------
// SList<T>::size
//------------------------------------------------------------------------
template <typename T>
int SList<T>::size() const {
  int n = 0;

  Node* curr_p = m_head_p;
  while (curr_p != nullptr) {
    n++;
    curr_p = curr_p->next_p;
  }

  return n;
  // END REMOVE
}

//------------------------------------------------------------------------
// SList<T>::at
//------------------------------------------------------------------------
template <typename T>
const T& SList<T>::at(int idx) const {
  Node* curr_p = m_head_p;
  for (int i = 0; i < idx; i++)
    curr_p = curr_p->next_p;

  return curr_p->value;
  // END REMOVE
}

//------------------------------------------------------------------------
// SList<T>::at
//------------------------------------------------------------------------
template <typename T>
T& SList<T>::at(int idx) {
  //''' LAB TASK '''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // Implement push_front
  //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // BEGIN REMOVE
  Node* curr_p = m_head_p;
  for (int i = 0; i < idx; i++)
    curr_p = curr_p->next_p;

  return curr_p->value;
  // END REMOVE
}
```

```
//------------------------------------------------------------------------
// SList Copy Constructor
//------------------------------------------------------------------------

template <typename T>
SList<T>::SList(const SList<T>& lst) {
  // BEGIN REMOVE
  // We must make sure head pointer is initialized correctly, otherwise
  // push_front will not work correctly.

  m_head_p = nullptr;

  // Iterate through each element of the given lst and use push_front to
  // add it to this list.

  Node* curr_p = lst.m_head_p;
  while (curr_p != nullptr) {
    push_front(curr_p->value);
    curr_p = curr_p->next_p;
  }

  // We now have all elements in this list, but they are in the reverse
  // order, so we can call reverse to ensure that this list is an exact
  // copy of the given list.

  reverse_v1();
  // END REMOVE
}

//------------------------------------------------------------------------
// SList Swap
//------------------------------------------------------------------------

template <typename T>
void SList<T>::swap(SList<T>& lst) {
  //''' LAB TASK ''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // Implement swap
  //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // BEGIN REMOVE
  ::swap(m_head_p, lst.m_head_p);
  // END REMOVE
}

//------------------------------------------------------------------------
// SList Overloaded Assignment Operator
//------------------------------------------------------------------------

template <typename T>
SList<T>& SList<T>::operator=(const SList<T>& lst) {
```

```
  //''' LAB TASK ''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // Implement operator=
  //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // BEGIN REMOVE
  SList<T> tmp(lst);  // create temporary copy of given list
  swap(tmp);          // swap this list with temporary list
  return *this;       // destructor called for temporary list
  // END REMOVE
}

//-----------------------------------------------------------------------
// SList<T>::reverse_v1
//-----------------------------------------------------------------------
// Pseudocode for this algorithm:
//
//  def reverse( x, n ):
//    for i in 0 to n/2:
//      swap( x[i], x[(n-1)-i] )
//

template <typename T>
void SList<T>::reverse_v1() {
  //''' LAB TASK ''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // Implement reverse_v1
  //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // here is one implementation of reverse. Instead of this, use swap.
  //   int lo = i;
  //   int hi = (n-1)-i;
  //
  //   int tmp = at(lo);
  //   at(lo)  = at(hi);
  //   at(hi)  = tmp;
  // BEGIN REMOVE
  // The implementation with the call to swap is faster than using the
  // above approach.
  //
  // This is because the above approach ends up traversing the once to
  // read lo, once to write lo, once to read hi, once to write hit. When
  // using the explicit call to swap we only to the traversal once and
  // then swap uses a non-const reference to both read and write the
  // values.

  int n = size();
  for (int i = 0; i < n / 2; i++)
    ::swap(at(i), at((n - 1) - i));
  // END REMOVE
}
```

```
//------------------------------------------------------------------------
// SList<T>::reverse_v2
//------------------------------------------------------------------------
// Steps for this algorithm:
//
//  1. Create temporary singly linked list
//  2. Push front all values from this list onto temporary list
//  3. Swap this list with the temporary list
//

template <typename T>
void SList<T>::reverse_v2() {
  //''' LAB TASK ''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // Implement reverse_v2
  //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // BEGIN REMOVE
  // Step 1. Create temporary list
  SList lst;

  // Step 2. Push front all values from this list onto temporary list
  Node* curr_p = m_head_p;
  while (curr_p != nullptr) {
    lst.push_front(curr_p->value);
    curr_p = curr_p->next_p;
  }

  // Step 3. Swap this list with temporary list
  swap(lst);
  // END REMOVE
}
```