```
//========================================================================
// 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 */
```

```
//------------------------------------------------------------------------
// SListInt Default Constructor
//------------------------------------------------------------------------
SListInt::SListInt() {
  //''' SECTION TASK #1 ''''''''''''''''''''''''''''''''''''''''''''''''''
  // Implement the constructor
  //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // BEGIN REMOVE
  m_head_p = nullptr;
  // END REMOVE
}

//------------------------------------------------------------------------
// SListInt Destructor
//------------------------------------------------------------------------
SListInt::~SListInt() {
  //''' SECTION TASK #1 ''''''''''''''''''''''''''''''''''''''''''''''''''
  // Delete the node
  //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // BEGIN REMOVE
  while ( m_head_p != nullptr ) {
    Node* temp_p = m_head_p->next_p;
    delete m_head_p;
    m_head_p = temp_p;
  }
  // END REMOVE
}

//------------------------------------------------------------------------
// SListInt::push_front
//------------------------------------------------------------------------
void SListInt::push_front(int v) {
  //''' SECTION TASK #1 ''''''''''''''''''''''''''''''''''''''''''''''''''
  // 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
}
```

```
//------------------------------------------------------------------------
// SListInt::size
//------------------------------------------------------------------------
int SListInt::size() const {
  //''' SECTION TASK #2''''''''''''''''''''''''''''''''''''''''''''''''''
  // Implement at
  //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // BEGIN REMOVE
  int n = 0;

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

  return n;
  // END REMOVE
}

//------------------------------------------------------------------------
// SListInt::at
//------------------------------------------------------------------------
int SListInt::at(int idx) const {
  //''' SECTION TASK #2''''''''''''''''''''''''''''''''''''''''''''''''''
  // Implement at
  //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // 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
}

//------------------------------------------------------------------------
// SListInt::at
//------------------------------------------------------------------------
int& SListInt::at(int idx) {
  //''' SECTION TASK #2''''''''''''''''''''''''''''''''''''''''''''''''''
  // Implement at
  //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // 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
}
```

```
//------------------------------------------------------------------------
// SListInt Copy Constructor
//------------------------------------------------------------------------
SListInt::SListInt(const SListInt& lst) {
  //''' SECTION TASK #3''''''''''''''''''''''''''''''''''''''''''''''''''
  // Implement at
  //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // BEGIN REMOVE
  m_head_p = nullptr;

  Node* curr_p = lst.m_head_p;
  while (curr_p != nullptr) {
    push_front(curr_p->value);
    curr_p = curr_p->next_p;
  }
  reverse_v1();
  // END REMOVE
}

//------------------------------------------------------------------------
// SListInt Swap
//------------------------------------------------------------------------
void SListInt::swap(SListInt& lst) {
  //''' SECTION TASK #3''''''''''''''''''''''''''''''''''''''''''''''''''
  // Implement at
  //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // BEGIN REMOVE
  Node* tmp_p  = m_head_p;
  m_head_p     = lst.m_head_p;
  lst.m_head_p = tmp_p;
  // END REMOVE
}

//------------------------------------------------------------------------
// SListInt Overloaded Assignment Operator
//------------------------------------------------------------------------
SListInt& SListInt::operator=(const SListInt& lst) {
  //''' SECTION TASK #3''''''''''''''''''''''''''''''''''''''''''''''''''
  // Implement at
  //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // BEGIN REMOVE
  SListInt tmp(lst);
  swap(tmp);
  return *this;
  // END REMOVE
}
```

```
//========================================================================
// types-dpoly.h
//========================================================================
// Declarations for misc types

#ifndef TYPES_DPOLY_H
#define TYPES_DPOLY_H

//------------------------------------------------------------------------
// IObject
//------------------------------------------------------------------------

class IObject {
 public:
  virtual ~IObject(){};
  virtual IObject* clone() const                = 0;
  virtual bool     eq(const IObject& rhs) const = 0;
  virtual bool     lt(const IObject& rhs) const = 0;
  virtual void     print() const                = 0;
};

bool operator==(const IObject& lhs, const IObject& rhs);
bool operator<(const IObject& lhs, const IObject& rhs);

//------------------------------------------------------------------------
// Integer
//------------------------------------------------------------------------

class Integer : public IObject {
 public:
  Integer();
  Integer(int data);

  Integer* clone() const;
  bool     eq(const IObject& rhs) const;
  bool     lt(const IObject& rhs) const;
  void     print() const;

 private:
  int m_data;
};

//------------------------------------------------------------------------
// Double
//------------------------------------------------------------------------

class Double : public IObject {
 public:
  Double();
  Double(double data);

  Double* clone() const;
  bool    eq(const IObject& rhs) const;
```

```
  bool    lt(const IObject& rhs) const;
  void    print() const;

 private:
  double m_data;
};

//------------------------------------------------------------------------
// Complex
//------------------------------------------------------------------------

class Complex : public IObject {
 public:
  Complex();
  Complex(double real, double imag);

  Complex* clone() const;
  bool     eq(const IObject& rhs) const;
  bool     lt(const IObject& rhs) const;
  void     print() const;

 private:
  double m_real;
  double m_imag;
};

#endif  // TYPES_DPOLY_H
```

```
//========================================================================
// types-dpoly.cc
//========================================================================
// Definitions for misc types

#include "types-dpoly.h"

#include <cstdio>

//------------------------------------------------------------------------
// Overloaded operators
//------------------------------------------------------------------------

bool operator==(const IObject& lhs, const IObject& rhs) {
  return lhs.eq(rhs);
}

bool operator<(const IObject& lhs, const IObject& rhs) {
  return lhs.lt(rhs);
}

//------------------------------------------------------------------------
// Integer
//------------------------------------------------------------------------

Integer::Integer() {
  m_data = 0;
}

Integer::Integer(int data) {
  m_data = data;
}

Integer* Integer::clone() const {
  //''' SECTION TASK ''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // Implement clone
  //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // BEGIN REMOVE
  return new Integer(*this);
  // END REMOVE




}

bool Integer::eq(const IObject& rhs) const {
  const Integer* rhs_p = dynamic_cast<const Integer*>(&rhs);
  if (rhs_p == nullptr)
    return false;
  else
    return (m_data == rhs_p->m_data);
```

```
}

bool Integer::lt(const IObject& rhs) const {
  const Integer* rhs_p = dynamic_cast<const Integer*>(&rhs);
  if (rhs_p == nullptr)
    return false;
  else
    return (m_data < rhs_p->m_data);
}

void Integer::print() const {
  std::printf("%d", m_data);
}

//------------------------------------------------------------------------
// Double
//------------------------------------------------------------------------

Double::Double() {
  m_data = 0.0;
}

Double::Double(double data) {
  m_data = data;
}

Double* Double::clone() const {
  //''' SECTION TASK '''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // Implement clone
  //'''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // BEGIN REMOVE
  return new Double(*this);
  // END REMOVE




}

bool Double::eq(const IObject& rhs) const {
  const Double* rhs_p = dynamic_cast<const Double*>(&rhs);
  if (rhs_p == nullptr)
    return false;
  else
    return (m_data == rhs_p->m_data);
}

bool Double::lt(const IObject& rhs) const {
  const Double* rhs_p = dynamic_cast<const Double*>(&rhs);
  if (rhs_p == nullptr)
    return false;
  else
    return (m_data < rhs_p->m_data);
```

```
}

void Double::print() const {
  std::printf("%f", m_data);
}

//------------------------------------------------------------------------
// Complex
//------------------------------------------------------------------------

Complex::Complex() {
  m_real = 0.0;
  m_imag = 0.0;
}

Complex::Complex(double real, double imag) {
  m_real = real;
  m_imag = imag;
}

Complex* Complex::clone() const {
  //''' SECTION TASK '''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // Implement clone
  //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // BEGIN REMOVE
  return new Complex(*this);
  // END REMOVE




}

bool Complex::eq(const IObject& rhs) const {
  const Complex* rhs_p = dynamic_cast<const Complex*>(&rhs);
  if (rhs_p == nullptr)
    return false;
  else
    return (m_real == rhs_p->m_real) && (m_imag == rhs_p->m_imag);
}

bool Complex::lt(const IObject& rhs) const {
  const Complex* rhs_p = dynamic_cast<const Complex*>(&rhs);
  if (rhs_p == nullptr)
    return false;
  else
    return (m_real < rhs_p->m_real) && (m_imag < rhs_p->m_imag);
}

void Complex::print() const {
  std::printf("%f+%fi", m_real, m_imag);
}
```

```
//========================================================================
// SListIObj.h
//========================================================================
// Declarations for SListIObj

#ifndef SLIST_IOBJ_H
#define SLIST_IOBJ_H

#include "types-dpoly.h"

class SListIObj {
  //----------------------------------------------------------------------
  // Constructor and destructor
  //----------------------------------------------------------------------
 public:
  SListIObj();
  ~SListIObj();

  //----------------------------------------------------------------------
  // Copy constructor, swap, assignment operator
  //----------------------------------------------------------------------
 public:
  SListIObj(const SListIObj& lst);
  void       swap(SListIObj& lst);
  SListIObj& operator=(const SListIObj& lst);

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

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

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

 private:
  struct Node {
    IObject* obj_p;
    Node*    next_p;
  };

  Node* m_head_p;
};

#endif /* SLIST_IOBJ_H */
```

```
//------------------------------------------------------------------------
// SListIObj Default Constructor
//------------------------------------------------------------------------
SListIObj::SListIObj() {
  //''' SECTION TASK #4 ''''''''''''''''''''''''''''''''''''''''''''''''''
  // Implement the constructor
  //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // BEGIN REMOVE
  m_head_p = nullptr;
  // END REMOVE
}

//------------------------------------------------------------------------
// SListIObj Destructor
//------------------------------------------------------------------------
SListIObj::~SListIObj() {
  //''' SECTION TASK #4 ''''''''''''''''''''''''''''''''''''''''''''''''''
  // Delete the node
  //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // BEGIN REMOVE
  while (m_head_p != nullptr) {
    Node* temp_p = m_head_p->next_p;
    delete m_head_p->obj_p;
    delete m_head_p;
    m_head_p = temp_p;
  }
  // END REMOVE
}

//------------------------------------------------------------------------
// SListIObj::push_front
//------------------------------------------------------------------------
void SListIObj::push_front(const IObject& v) {
  //''' SECTION TASK #4 ''''''''''''''''''''''''''''''''''''''''''''''''''
  // push to the front
  //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // BEGIN REMOVE
  Node* new_node_p   = new Node;
  new_node_p->obj_p  = v.clone();
  new_node_p->next_p = m_head_p;
  m_head_p           = new_node_p;
  // END REMOVE
}
```

```
//------------------------------------------------------------------------
// SListIObj::size
//------------------------------------------------------------------------
int SListIObj::size() const {
  //''' SECTION TASK #5 ''''''''''''''''''''''''''''''''''''''''''''''''''
  // Implement the constructor
  //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // BEGIN REMOVE
  int n = 0;

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

  return n;
  // END REMOVE
}

//------------------------------------------------------------------------
// SListIObj::at
//------------------------------------------------------------------------
IObject* SListIObj::at(int idx) const {
  //''' SECTION TASK #5 ''''''''''''''''''''''''''''''''''''''''''''''''''
  // Implement the constructor
  //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // BEGIN REMOVE
  Node* curr_p = m_head_p;
  for (int i = 0; i < idx; i++)
    curr_p = curr_p->next_p;

  return curr_p->obj_p;
  // END REMOVE
}

//------------------------------------------------------------------------
// SListIObj::at
//------------------------------------------------------------------------
IObject*& SListIObj::at(int idx) {
  //''' SECTION TASK #5 ''''''''''''''''''''''''''''''''''''''''''''''''''
  // Implement the constructor
  //''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
  // BEGIN REMOVE
  Node* curr_p = m_head_p;
  for (int i = 0; i < idx; i++)
    curr_p = curr_p->next_p;

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