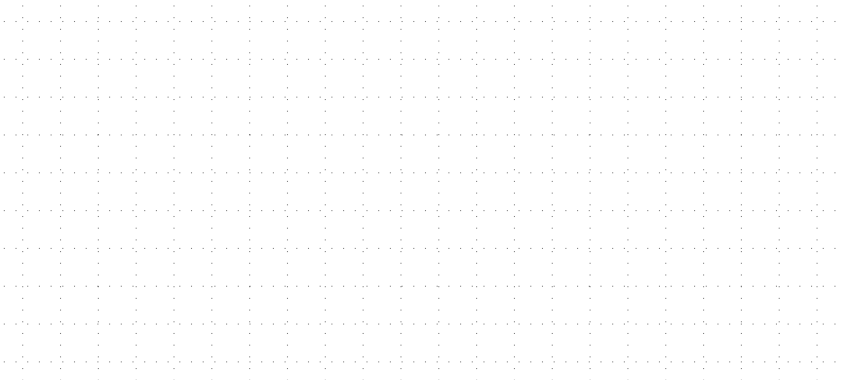
3. C++ Inheritance

What do we think of this?

```
class Pawn
{
 public:

  Pawn( char col, int row )
  {
    m_col = col;
    m_row = row;
  }

  int get_pts() const
  {
    return 1;
  }

  char get_col() const
  {
    return m_col;
  }

  int get_row() const
  {
    return m_row;
  }

  void move( char col, int row )
  {
    if (    (col != m_col)
         || (row != (m_row + 1)) )
     throw 42;

    m_col = col;
    m_row = row;
  }

 private:
  char m_col;
  int  m_row;
};
```

```
class Rook
{
 public:

  Rook( char col, int row )
  {
    m_col = col;
    m_row = row;
  }

  int get_pts() const
  {
    return 5;
  }

  char get_col() const
  {
    return m_col;
  }

  int get_row() const
  {
    return m_row;
  }

  void move( char col, int row )
  {
    if (    (col != m_col)
         && (row != m_row) )
     throw 42;

    m_col = col;
    m_row = row;
  }

 private:
  char m_col;
  int  m_row;
};
```

- Object-oriented design without inheritance



- We want to be able to ...

  ... create algorithms and data structures that work for all pieces

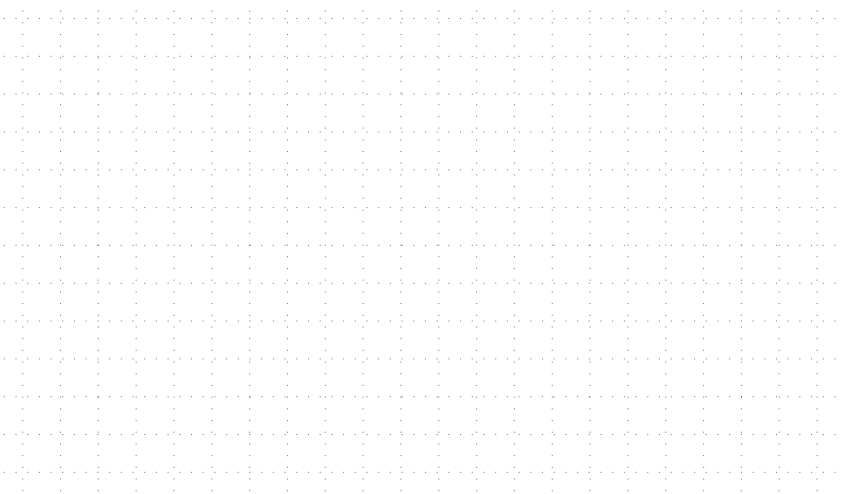
  ... easily reuse implementation code across pieces

- Inheritance enables declaring a derived class that automatically includes the interface and implementation of a different base class
  - Derived class also called the child class or subclass
  - Base class also called the parent class or superclass

- We will take an incremental approach
  - Implementation inheritance
  - From implementation to interface inheritance
  - Interface inheritance

3.1. C++ Implementation Inheritance

- Object-oriented design with implementation inheritance

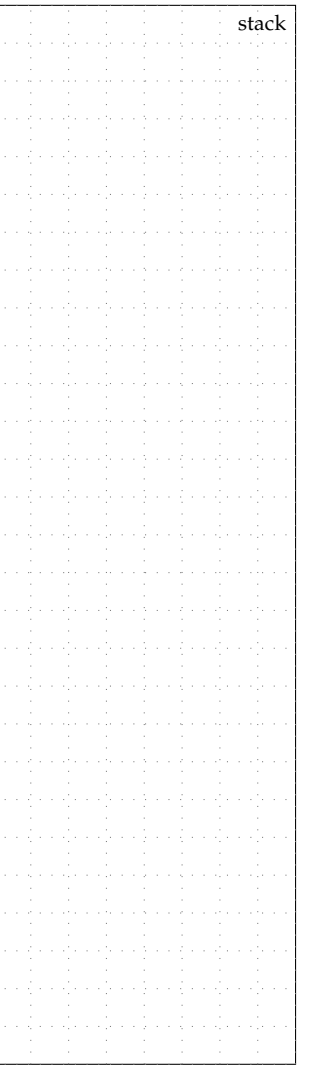


	Public Inheritance	**Protected Inheritance**	**Private Inheritance**
Public members in base class	**Public** in derived class	**Protected** in derived class	**Private** in derived class
Protected members in base class	**Protected** in derived class	**Protected** in derived class	**Private** in derived class
Private members in base class	**Not accessible** in derived class	**Not accessible** in derived class	**Not accessible** in derived class

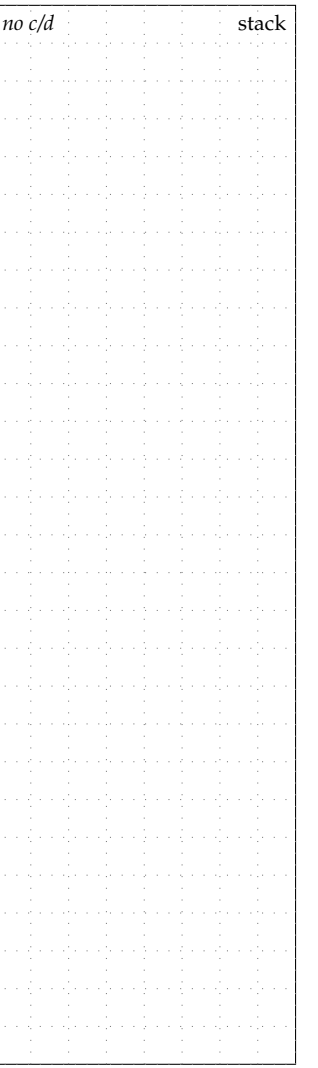
```
class Piece
{
 public:

  Piece( char col, int row )
  {
    m_col = col;
    m_row = row;
  }

  char get_col() const
  {
   return m_col;
  }

  int get_row() const
  {
   return m_row;
  }









  void move( char col, int row )
  {
    m_col = col;
    m_row = row;
  }




 // derived classes cannot
 // access private data in
 // base class
 protected:
  char m_col;
  int  m_row;

};
```

```
class Rook : public Piece
{
 public:

  Rook( char col, int row )
   : Piece( col, row )
  { }









  int get_pts() const
  {
    return 5;
  }

  void move( char col, int row )
  {
    if (    (col != m_col)
         && (row != m_row) )
      throw 42;

    Piece::move( col, row );
  }




};
```

```
01 class Piece
02 {
03   Piece( char col, int row )
04   {
05     m_col = col;
06     m_row = row;
07   }
08
09   char get_col() const {
10     return m_col;
11   }
12
13   int get_row() const {
14     return m_row;
15   }
16   ...
17 };
18
19 class Pawn : public Piece
20 {
21   Pawn( char col, int row )
22    : Piece( col, row )
23   { }
24
25   int get_pts() const {
26     return 1;
27   }
28   ...
29 };
30
31 int main( void )
32 {
33   Pawn p('a',2);
34   char col = p.get_col();
35   return 0;
36 }
```



```
01 class Piece
02 {
03   Piece( char col, int row ) {
04     m_col = col;
05     m_row = row;
06   }
07
08   char get_col() const {
09     return m_col;
10   }
11
12   int get_row() const {
13     return m_row;
14   }
15   ...
16 };
17
18 class Pawn : public Piece
19 {
20   int get_pts() const { return 1; }
21   ...
22 };
23
24 class Rook : public Piece
25 {
26   int get_pts() const { return 5; }
27   ...
28 };
29
30 int main( void )
31 {
32   Pawn p('a',2);
33   Rook r('h',3);
34   int pts0 = p.get_pts();
35   int pts1 = r.get_pts();
36   int sum  = pts0 + pts1;
37   return 0;
38 }
```



3.2. From Implementation to Interface Inheritance

- Implementation Inheritance
  - really more of the composition relationship ("has a")
  - focuses on refactoring implementation code into the base class
  - still only instantiate and manipulate derived "implementation" classes
  - need to know all type information at compile time
- Interface Inheritance
  - the key to leveraging the generalization relationship ("is a")
  - focuses on enabling dynamic polymorphism (i.e., subtype polymorphism)
  - can manipulate objects using the base "interface" class
  - do not need to know implementation type information at compile time
  - can use run-time type information

Type conversion and casting in class hierarchies

```
1  Rook rook('h',3);
2
3  // implicit type conversion
4  Piece* piece_p = &rook;
5
6  // can only call methods defined in Piece
7  piece_p->get_col();
8
9  // explicit type casting to correct type
10 Rook* rook_p = (Rook*) piece_p;
11
12 // can call methods defined in Piece or Rook
13 rook_p->get_pts();
```

Dynamic Polymorphism

- Consider the following code snippet from previous example

```
1 int pts0 = p.get_pts();
2 int pts1 = r.get_pts();
3 int sum  = pts0 + pts1;
```

- Consider refactoring this code into a separate function
  - we want our function to be able to work with any type of chess piece
  - this function exhibits dynamic polymorphism
  - polymorphism = condition of occurring in several different forms
  - dynamic = run-time

```
1 int calc_pts( Piece* p0, Piece* p1 )
2 {
3   int pts0 = p0->get_pts();
4   int pts1 = p1->get_pts();
5   return pts0 + pts1;
6 }
```

Possible but rejected approaches to dynamic dispatch:

- Cast in the caller
  - check type field in every function that needs to call a derived method (e.g., calc_pts)
  - cast the base pointer to the right derived type
  - call the method Works, but every caller must know all derived types and the logic duplicates everywhere.
- Cast in the base class (move the type-checking logic into a base class method)
  - (Piece::get_pts) checks its own type field
  - casts this to correct sub-type
  - calls the method Centralizes the dispatch, but the base class now has to know about all derived classes

These solutions are not only inelegant, but technically problematic!

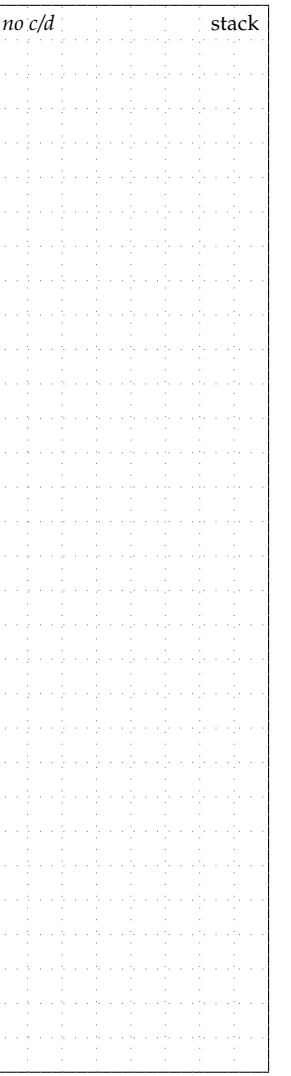
3.3. C++ Interface Inheritance

- C++ includes language support for dynamic dispatch
- `virtual` keyword can be used with a base class member function
- Calling a virtual member function will dynamically dispatch to the appropriate derived class member function
- At least one virtual function, compiler generates implicit type field
- Compiler generates more optimized version of dynamic dispatch

```
class Piece
{
 public:

  Piece( char col, int row )
  {
    m_col = col;
    m_row = row;
  }

  virtual int get_pts() const = 0;

  char get_col() const
  {
    return m_col;
  }

  int get_row() const
  {
    return m_row;
  }

  void move( char col, int row )
  {
    m_col = col;
    m_row = row;
  }

 protected:
  char m_col;
  int  m_row;
};
```

```
class Rook : public Piece
{
 public:

  Rook( char col, int row )
   : Piece( col, row )
  { }

  int get_pts() const
  {
    return 5;
  }

  void move( char col, int row )
  {
    if (    (col != m_col)
         && (row != m_row) )
     throw 42;

    Piece::move( col, row );
  }

};
```

```
01 class Piece
02 {
03   virtual int get_pts() const = 0;
04   ...
05 };
06
07 class Pawn : public Piece
08 {
09   int get_pts() const { return 1; }
10   ...
11 };
12
13 class Rook : public Piece
14 {
15   int get_pts() const { return 5; }
16   ...
17 };
18
19 int calc_pts( Piece* p0, Piece* p1 )
20 {
21   int pts0 = p0->get_pts();
22   int pts1 = p1->get_pts();
23   return pts0 + pts1;
24 }
25
26 int main( void )
27 {
28   Pawn p('a',2);
29   Rook r('h',3);
30   int sum = calc_pts( &p, &r );
31   return 0;
32 }
```

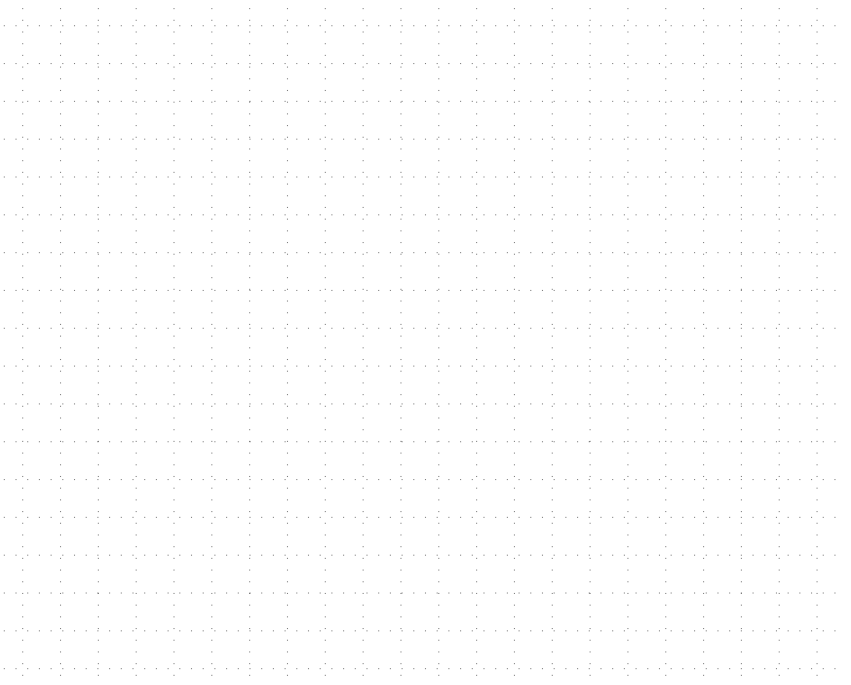


Abstract base classes

- All member functions are pure virtual member functions
- Need to have a non-pure virtual destructor in base class

```
1  class IPiece
2  {
3   public:
4    virtual ~IPiece() { }
5    virtual int  get_pts() const = 0;
6    virtual char get_col() const = 0;
7    virtual int  get_row() const = 0;
8    virtual void move( char col, int row ) = 0;
9  };
10
11 class Rook : public IPiece
12 {
13  public:
14   Rook( char col, int row )
15   {
16     m_col = col;
17     m_row = row;
18   }
19
20   ~Rook() { }
21
22   int  get_pts() const { return 5;     }
23   char get_col() const { return m_col; }
24   int  get_row() const { return m_row; }
25
26   void move( char col, int row )
27   {
28     if ( (col != m_col) && (row != m_row) )
29       throw 42;
30     m_col = col;
31     m_row = row;
32   }
33
34  private:
35   char m_col;
36   int  m_row;
37 };
```

3.4. Revisiting Composition vs. Generalization



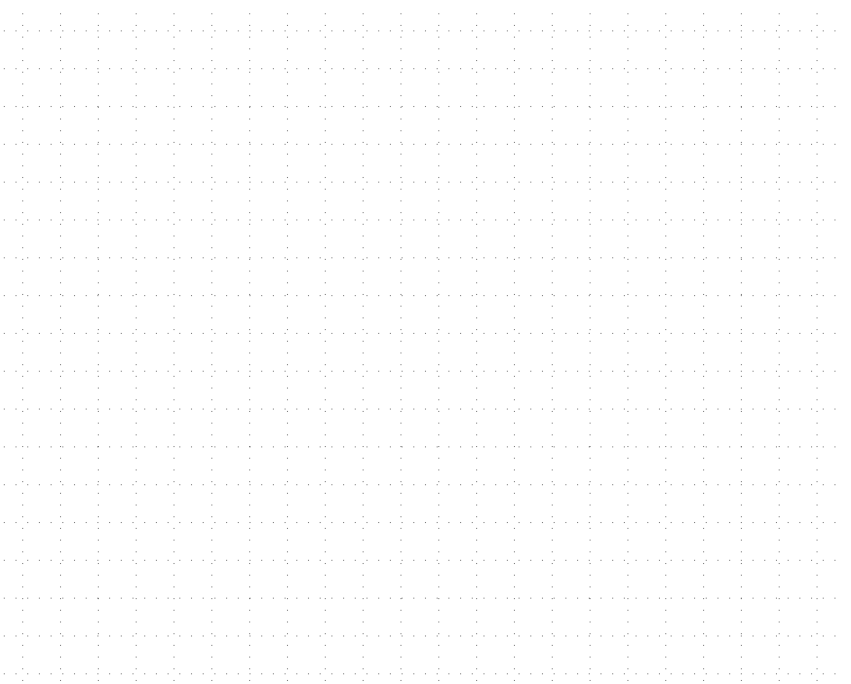
```
class Position
{
 public:

  Position()
  {
    m_col = '?';
    m_row = 0;
  }

  Position( char col, int row )
  {
    m_col = col;
    m_row = row;
  }

  char get_col() const
  {
    return m_col;
  }

  int get_row() const
  {
    return m_row;
  }

  void move( char col, int row )
  {
    m_col = col;
    m_row = row;
  }

 private:
  char m_col;
  int  m_row;

};
```

```
class Pawn : public IPiece
{
 public:

  Pawn( char col, int row )
  {
    m_pos = Position(col,row);
  }

  ~Pawn()
  { }

  int get_pts() const
  {
    return 1;
  }

  char get_col() const
  {
    return m_pos.get_col();
  }

  int get_row() const
  {
    return m_pos.get_row();
  }

  void move( char col, int row )
  {
    int curr_col = m_pos.get_col();
    int curr_row = m_pos.get_row();
    if (    (col != curr_col)
         || (row != (curr_row + 1)) )
      throw 42;
    m_pos.move( col, row );
  }

 private:
  Position m_pos;
};
```

4. Object-Oriented Data Structures with Dynamic Polymorphism

- So far our list data structure can only store `ints`
- If we want a list of doubles → copy-and-paste
- If we want a list of complex numbers → copy-and-paste
- We have no way to store elements of different types in a list
- We can use dynamic polymorphism to create a polymorphic list

Class Hierarchy for Objects



```
class IObject
{
 public:

  virtual ~IObject() { };
  virtual IObject*    clone( )                 const = 0;
  virtual bool        eq( const IObject& obj ) const = 0;
  virtual bool        lt( const IObject& obj ) const = 0;

};

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

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

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

```
class Integer : public IObject
{
 public:
  Integer()        { m_i = 0; }
  Integer( int i ) { m_i = i; }

  Integer* clone() const
  {
    return new Integer( *this );
  }

  bool eq( const IObject& obj ) const
  {
    const Integer* int_p = dynamic_cast<const Integer*>( &obj );
    if ( int_p == nullptr )
      return false;
    else
      return ( m_i == int_p->m_i );
  }

  bool lt( const IObject& obj ) const
  {
    const Integer* int_p = dynamic_cast<const Integer*>( &obj );
    if ( int_p == nullptr )
      return false;
    else
      return ( m_i < int_p->m_i );
  }

 private:
  int m_i;
};
```

```
01 class IObject
02 {
03  public:
04    virtual bool
05    eq( const IObject& obj ) const = 0;
06   ...
07 };
08
09 bool operator==( const IObject& lhs,
10                  const IObject& rhs ) {
11   return lhs.eq(rhs);
12 }
13
14 class Integer : public IObject
15 {
16  public:
17   Integer( int data ) { m_i = i }
18
19   bool eq( const IObject& obj ) const {
20    const Integer* int_p
21     = dynamic_cast<const Integer*>(&obj);
22    if ( int_p == nullptr )
23      return false;
24    else
25      return ( m_i == int_p->m_i );
26   }
27   ...
28  private:
29   int m_i;
30 };
31
32 int main( void )
33 {
34   Integer a(2);
35   Integer b(3);
36   bool c = ( a == b );
37   return 0;
38 }
```

stack

4.1. Singly Linked List Interface

- Object-oriented list which stores `int`s

```
class SListInt
{
 public:
  SListInt();
  ~SListInt();
  void push_front(
    int v );
  ...

  class Itr
  {
   public:
    Itr( Node* node_p );
    void next();
    int& get();
    bool eq( Itr itr ) const;

   private:
    Node* m_node_p;
  };

  Itr begin();
  Itr end();

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

  Node* m_head_p;
};
```

- Object-oriented list which stores `IObject`s

```
class SListIObj
{
 public:
  SListIObj();
  ~SListIObj();
  void push_front(
    const IObject& v );
  ...

  class Itr
  {
   public:
    Itr( Node* node_p );
    void next();
    IObject*& get();
    bool eq( Itr itr ) const;

   private:
    Node* m_node_p;
  };

  Itr begin();
  Itr end();

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

  Node* m_head_p;
};
```

4.2. Singly Linked List Implementation

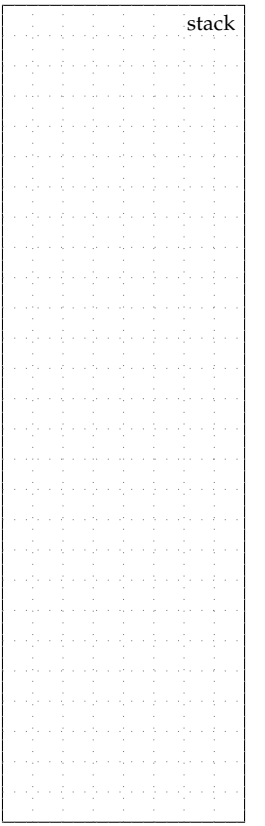
- Implementation for singly linked list to store `ints`

- Implementation for singly linked list to store `IObjects`

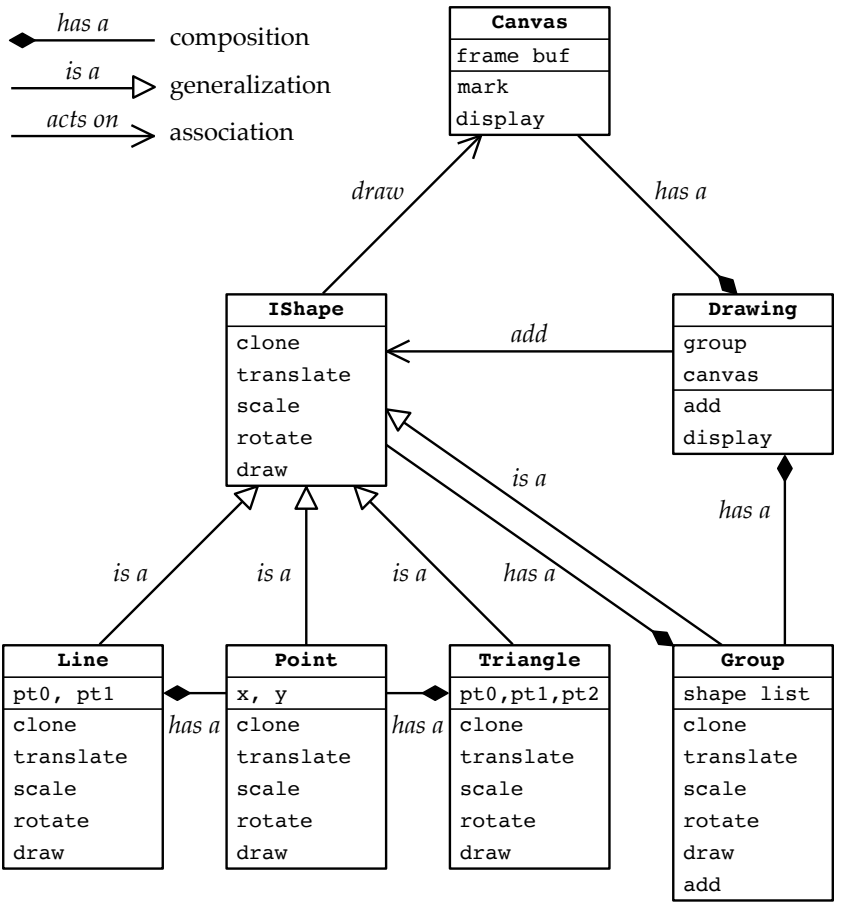
```
SListInt::SListInt()
{
  m_head_p = nullptr;
}

void SListInt::push_front( int v )

{
  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;
}

SListInt::~SListInt()
{
  while ( m_head_p != nullptr ) {
    Node* temp_p = m_head_p->next_p;

    delete m_head_p;
    m_head_p = temp_p;
  }
}
```

```
SListIObj::SListIObj()
{
  m_head_p = nullptr;
}

void SListIObj::push_front(
                const IObject& v )
{
  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;
}

SListIObj::~SListIObj()
{
  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;
  }
}
```

```
01 class IObject
02 {
03  public:
04    virtual IObject* clone( ) const = 0;
05   ...
06 };
07
08 class Integer : public IObject
09 {
10  public:
11   Integer( int data ) { m_i = i; }
12   Integer* clone() const {
13     return new Integer( *this );
14   }
15   ...
16  private:
17   int m_i;
18 };
19
20 SListIObj::SListIObj() {
21   m_head_p = nullptr;
22 }
23
24 void SListIObj::push_front(
25                 const IObject& v ) {
26   Node* new_node_p   = new Node;
27   new_node_p->obj_p  = v.clone();
28   new_node_p->next_p = m_head_p;
29   m_head_p           = new_node_p;
30 }
31
32 int main( void )
33 {
34   SListIObj list;
35   Integer a(12);
36   list.push_front(a);
37   return 0;
38 }
```



5. Drawing Framework Case Study



Storing shapes using a statically sized array of `IShape` pointers

- Dynamic polymorphism to clone, transform, and draw shapes
- `Group` handles all of the dynamic memory management

```
class Group : public IShape
{
 public:
  ...

  ~Group() {
    for ( int i = 0; i < m_shapes_size; i++ )
      delete m_shapes[i];
  }

  void add( const IShape& shape ) {
    assert( m_shapes_size < 16 );
    m_shapes[m_shapes_size] = shape.clone();
    m_shapes_size++;
  }

  void translate( double x_offset, double y_offset ) {
    for ( int i = 0; i < m_shapes_size; i++ )
      m_shapes[i]->translate( x_offset, y_offset );
  }

  void draw( Canvas* canvas ) const {
    for ( int i = 0; i < m_shapes_size; i++ )
      m_shapes[i]->draw( canvas );
  }

 private:
  int     m_shapes_size;
  IShape* m_shapes[16];
};
```

Storing shapes using a dynamic polymorphic list

- Modify `IShape` to inherit from `IObject`
- `Group` does not handle any of the dynamic memory management

```
1  class Group : public IShape
2  {
3   public:
4    ...
5
6    ~Group()
7    { }
8
9    void add( const IShape& shape )
10   {
11     m_shapes.push_front( shape );
12   }
13
14   void translate( double x_offset, double y_offset )
15   {
16     for ( IObject* obj_p : m_shapes ) {
17       IShape* shape_p = dynamic_cast<IShape*>(obj_p);
18       shape->translate( x_offset, y_offset );
19     }
20   }
21
22   void draw( Canvas* canvas ) const
23   {
24     for ( IObject* obj_p : m_shapes ) {
25       IShape* shape_p = dynamic_cast<IShape*>(obj_p);
26       shape->draw( canvas );
27     }
28   }
29
30  private:
31   SListIObj m_shapes;
32 };
```