ECE 2400 Computer Systems Programming Fall 2017

Topic 14: C++ Inheritance

School of Electrical and Computer Engineering
Cornell University

revision: 2017-11-06-09-36

1 Introduction to C++ Inheritance **2**

1.1. Object-Oriented Design Without Inheritance 2
1.2. Inheriting Member Fields and Functions 4
1.3. Method Overriding 7
1.4. Dynamic Polymorphism via Virtual Methods 11
1.5. Abstract Base Classes 14
1.6. Revisiting Composition vs. Generalization 16

2 List Using Dynamic Polymorphism **18**

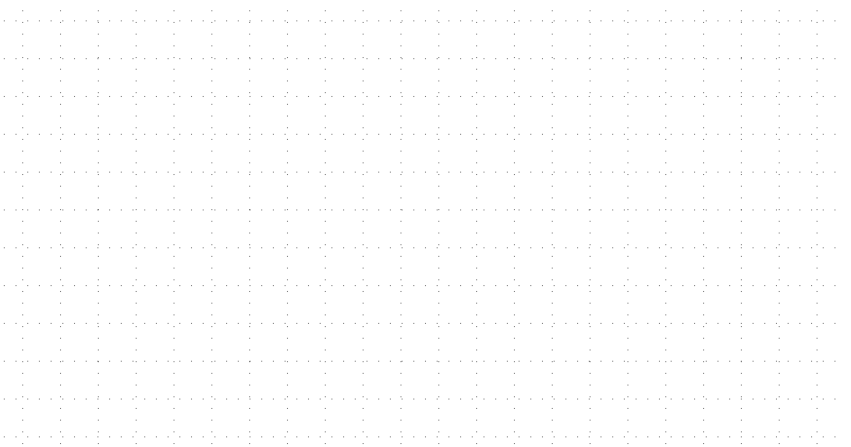
2.1. Class Hierarchy for Objects 18
2.2. Basic List Interface and Implementation 22
2.3. Iterator-Based List Interface and Implementation 25
2.4. Sorting with Iterators 28

3 Drawing Framework Using Dynamic Polymorphism **30**

1. Introduction to C++ Inheritance

- Inheritance enables declaring a derived class such that it automatically includes all of the member fields and functions associated with a different base class
  - Derived class also called the child class or subclass
  - Base class also called the parent class or superclass

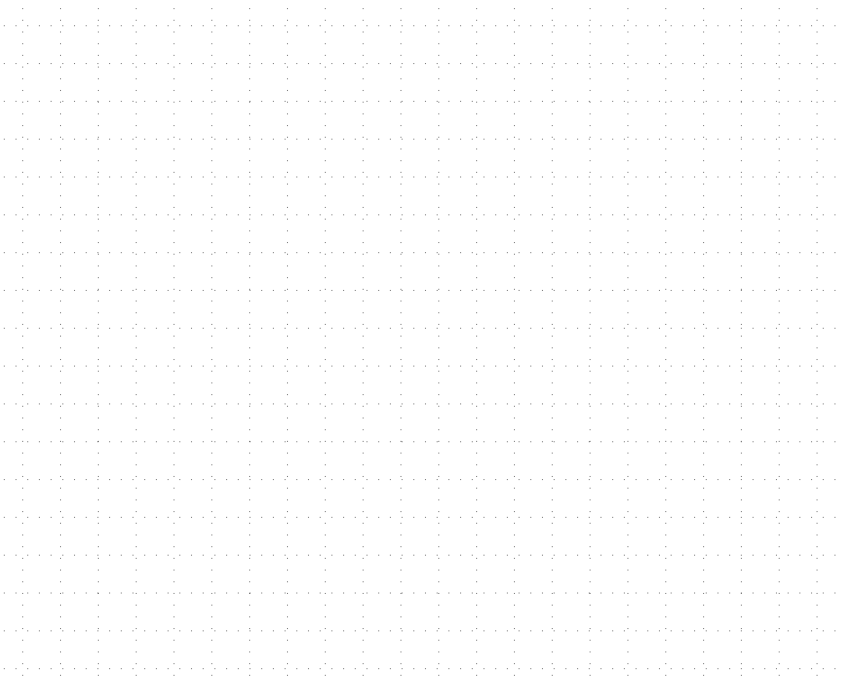
1.1. Object-Oriented Design Without Inheritance



```
class Pawn
{
 public:

  Pawn( 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 )
  {
   if (    (col != m_col)
        || (row != (m_row + 1)) )
    throw std::invalid_argument();

   m_col = col;
   m_row = row;
  }

  std::string to_str() const
  {
   std::stringstream ss;
   ss << "pawn@" << m_col << m_row;
   return ss.str();
  }

 private:
  char m_col;
  int  m_row;

};
```

```
class Rook
{
 public:

  Rook( 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 )
  {
   if (    (col != m_col)
        && (row != m_row) )
    throw std::invalid_argument();

   m_col = col;
   m_row = row;
  }

  std::string to_str() const
  {
   std::stringstream ss;
   ss << "pawn@" << m_col << m_row;
   return ss.str();
  }

 private:
  char m_col;
  int  m_row;

};
```

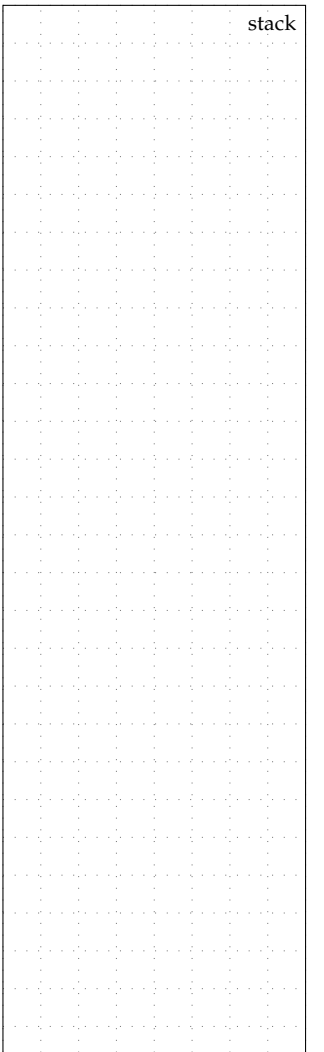
1.2. Inheriting Member Fields and Functions



```
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;
  }

 // derived classes cannot
 // access private data in
 // base class
 protected:

  char m_col;
  int  m_row;

};
```

```
class Pawn : public Piece
{
 public:

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

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

   m_col = col;
   m_row = row;
  }

  std::string to_str() const
  {
   std::stringstream ss;
   ss << "pawn@" << m_col << m_row;
   return ss.str();
  }

};
```

```
1 int main( void )
2 {
3   Pawn pawn( 'a', 2 );
4   int row = pawn.get_row();
5   pawn.move( 'a', 3 );
6   return 0;
7 }
```

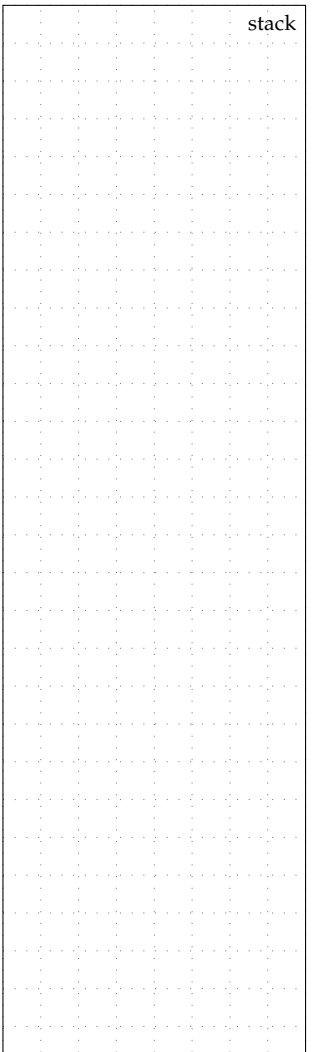


1.3. Method Overriding

```
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;
  }

  std::string to_str() const
  {
   std::stringstream ss;
   ss << m_col << m_row;
   return ss.str();
  }

 protected:

  char m_col;
  int  m_row;

};
```

```
class Pawn : public Piece
{
 public:

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

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

   Piece::move( col, row );
  }

  std::string to_str() const
  {
   std::stringstream ss;
   ss << "pawn@"
      << Piece::to_str( col, row );
   return ss.str();
  }

};
```

```
1  // polymorphic function?
2  void move_piece( Piece* p,
3                   char col, int row )
4  {
5    using namespace std;
6    cout << p->to_str() << " to ";
7    p->move( row, col );
8    cout << p->to_str() << endl;
9  }
10
11 int main( void )
12 {
13   Pawn pawn( 'a', 2 );
14   move_piece( &pawn, 'a', 3 );
15   Rook rook( 'h', 1 );
16   move_piece( &rook, 'h', 5 );
17   return 0;
18 }
```



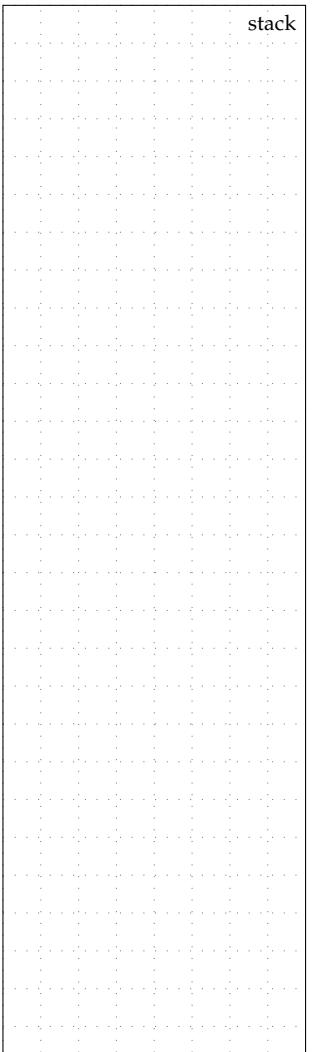
- Slicing is when we loose information by accessing a base class via the derived class
- Slicing can occur when we
  - Copy derived class to variable of the base class type
  - Access derived class via a variable of the base class pointer type

- We need a way to determine the concrete type of the object pointed to by a `Piece*` so we can cast the `Piece*` pointer to either a `Pawn*` or `Rook*` pointer
- We could add a new `m_type` field to `Piece`

```
enum PieceType { PAWN, ROOK };

class Piece
{
 public:

  Piece( PieceType type, char col, int row )
    : m_type(type), m_col(col), m_row(row)
  { }

  PieceType get_type() const
  {
    return m_type;
  }

  ...

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

```
class Pawn : public Piece
{
 public:

  Pawn( char col, int row )
    : Piece( PAWN, col, row )
  { }

  ...
};

class Rook : public Piece
{
 public:

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

  ...
};
```

```
1  // polymorphic function?
2  void move_piece( Piece* p,
3                   char row, int col )
4  {
5    using namespace std;
6    if ( p->get_type() == PAWN ) {
7      Pawn* q = (Pawn*) p;
8      cout << q->to_str() << " to ";
9      q->move( row, col );
10     cout << q->to_str() << endl;
11   }
12   else if ( p->get_type() == ROOK ) {
13     Rook* q = (Rook*) p;
14     cout << q->to_str() << " to ";
15     q->move( row, col );
16     cout << q->to_str() << endl;
17   }
18 }
19
20 int main( void )
21 {
22   Pawn pawn( 'a', 2 );
23   move_piece( &pawn, 'a', 3 );
24   Rook rook( 'h', 1 );
25   move_piece( &rook, 'h', 5 );
26   return 0;
27 }
```

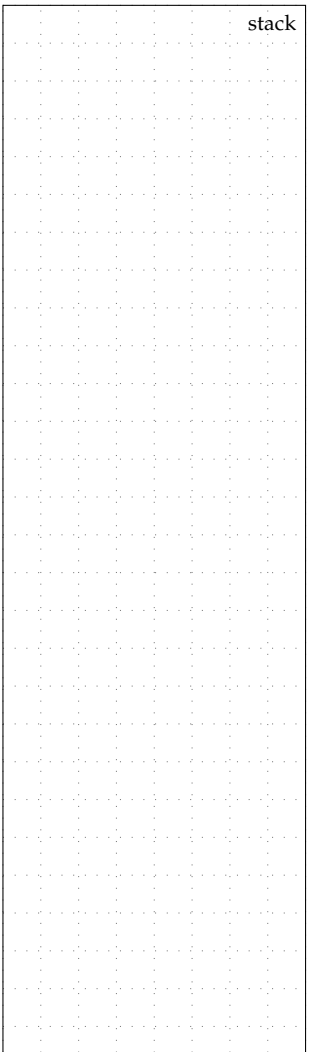


1.4. Dynamic Polymorphism via Virtual Methods

```
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;
  }

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

  virtual
  std::string to_str() const
  {
   std::stringstream ss;
   ss << m_col << m_row;
   return ss.str();
  }

 protected:

  char m_col;
  int  m_row;

};
```

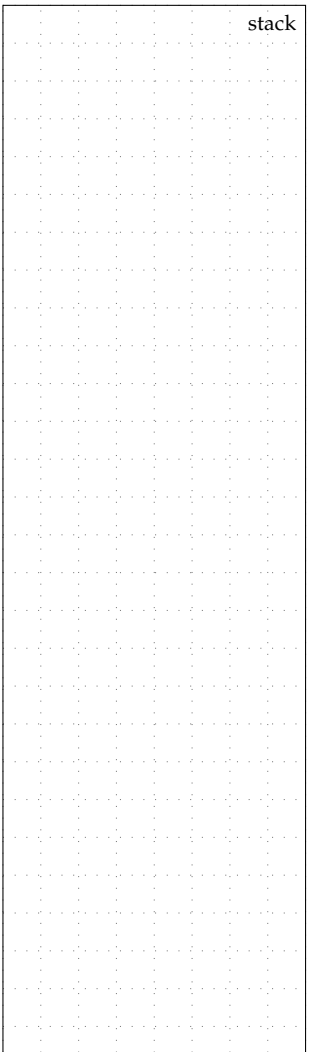
```
class Pawn : public Piece
{
 public:

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

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

   Piece::move( col, row );
  }

  std::string to_str() const
  {
   std::stringstream ss;
   ss << "pawn@"
      << Piece::to_str( col, row );
   return ss.str();
  }

};
```

```
1  // polymorphic function!
2  void move_piece( Piece* p,
3                   char col, int row )
4  {
5    using namespace std;
6    cout << p->to_str() << " to ";
7    p->move( row, col );
8    cout << p->to_str() << endl;
9  }
10
11 int main( void )
12 {
13   Pawn pawn( 'a', 2 );
14   move_piece( &pawn, 'a', 3 );
15   Rook rook( 'h', 1 );
16   move_piece( &rook, 'h', 5 );
17   return 0;
18 }
```



- Dynamic polymporphism enables a single interface (e.g., function) that operates over many types, where the type information is not known until runtime

```
1  // polymorphic function!
2  void move_piece( Piece* p,
3                   char col, int row )
4  {
5    using namespace std;
6    cout << p->to_str() << " to ";
7    p->move( row, col );
8    cout << p->to_str() << endl;
9  }
10
11 int main( void )
12 {
13   Pawn pawn( 'a', 2 );
14   Piece piece = pawn;
15   move_piece( &piece, 'a', 3 );
16   return 0;
17 }
```

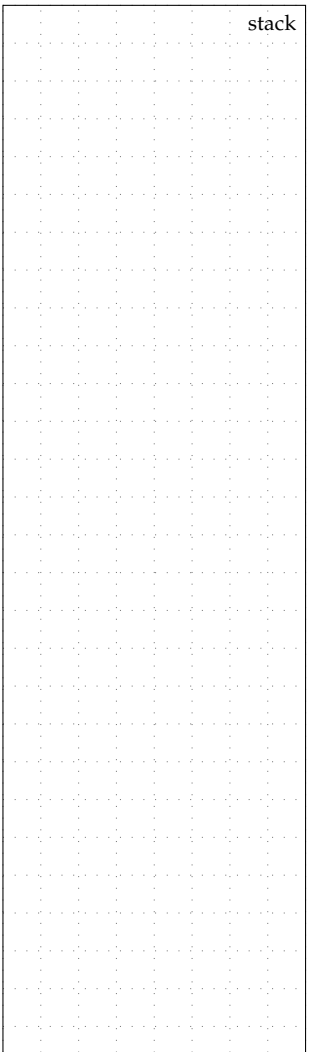


1.5. Abstract Base Classes

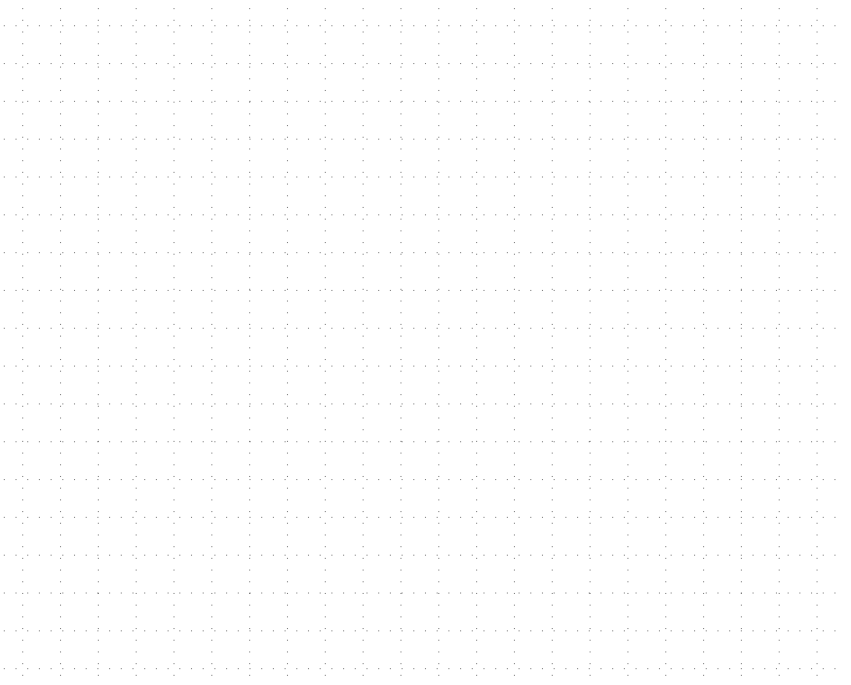
```
class IPiece
{
 public:

  virtual
  ~IPiece() { };

  virtual
  char get_col() const = 0;

  virtual
  int get_row() const = 0;

  virtual
  void move( char col, int row ) = 0;

  virtual
  std::string to_str() const = 0;

};
```

```
class Pawn : public IPiece
{
 public:

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

  ~Pawn()
  { }

  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 std::invalid_argument();

   m_col = col;
   m_row = row;
  }

  std::string to_str() const
  {
   std::stringstream ss;
   ss << "pawn@" << m_col << m_row;
   return ss.str();
  }

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

```
1  // polymorphic function!
2  void move_piece( Piece* p,
3                   char col, int row )
4  {
5    using namespace std;
6    cout << p->to_str() << " to ";
7    p->move( row, col );
8    cout << p->to_str() << endl;
9  }
10
11 int main( void )
12 {
13   Pawn pawn( 'a', 2 );
14   move_piece( &pawn, 'a', 3 );
15   Rook rook( 'h', 1 );
16   move_piece( &rook, 'h', 5 );
17   return 0;
18 }
```



1.6. Revisiting Composition vs. Generalization



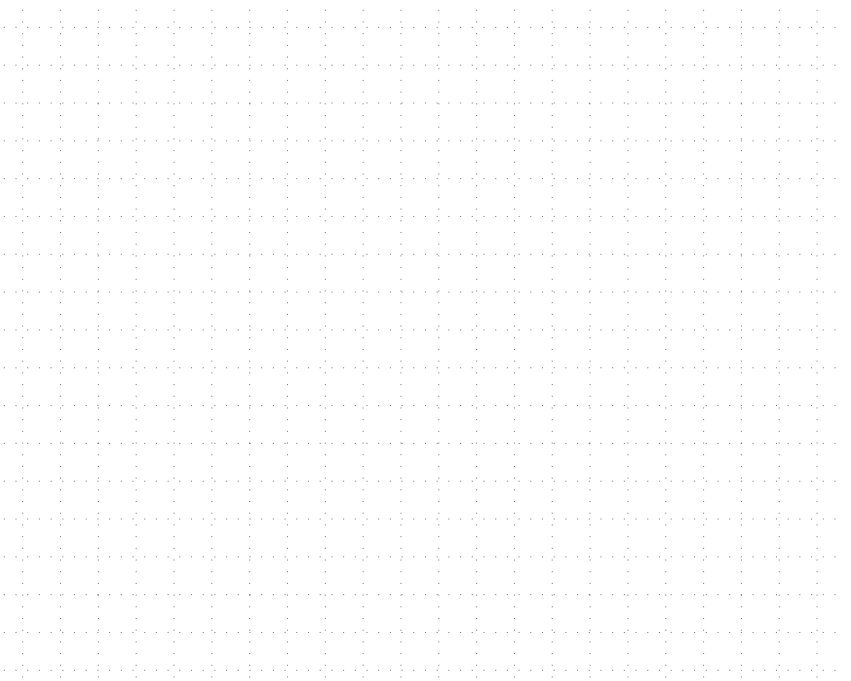
```
class Position
{
 public:
  Position( char col, int row )
    : m_col(col), m_row(row)
  { }

  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;
  }

  std::string to_str() const
  {
    std::stringstream ss;
    ss << m_col << m_row;
    return ss.str();
  }

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

```
class Pawn : public IPiece
{
 public:

  Pawn( char col, int row )
    : m_pos(col,row)
  { }

  ~Pawn()
  { }

  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 std::invalid_argument();

    m_pos.move( col, row );
  }

  std::string to_str() const
  {
    std::stringstream ss;
    ss << "pawn@" << m_pos.to_str();
    return ss.str();
  }

 private:
  Position m_pos;
};
```

2. List Using 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

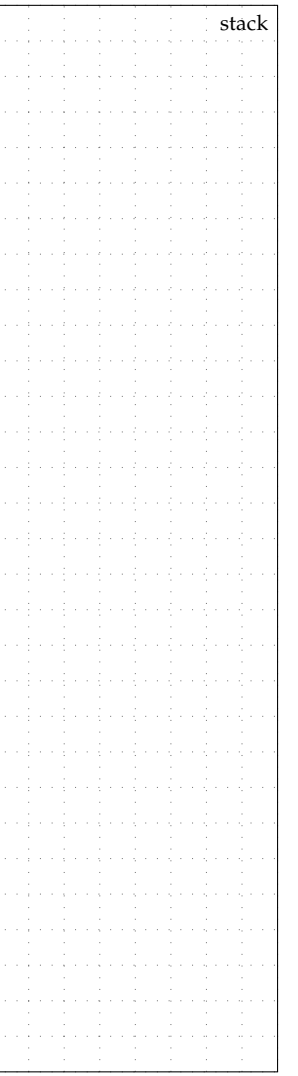
2.1. Class Hierarchy for Objects



```
1  class IObject
2  {
3   public:
4
5    virtual ~IObject()
6    { };
7
8    virtual IObject*    clone( )                  const = 0;
9    virtual std::string to_str()                  const = 0;
10   virtual bool        eq( const IObject& rhs )  const = 0;
11   virtual bool        lt( const IObject& rhs )  const = 0;
12
13 };
14
15 bool operator==( const IObject& lhs, const IObject& rhs )
16 {
17   return lhs.eq(rhs);
18 }
19
20 bool operator!=( const IObject& lhs, const IObject& rhs )
21 {
22   return !lhs.eq(rhs);
23 }
24
25 bool operator<( const IObject& lhs, const IObject& rhs )
26 {
27   return lhs.lt(rhs);
28 }
```

```
1  class Integer : public IObject
2  {
3   public:
4    Integer()                    : m_data(0)        { }
5    Integer( int data )          : m_data(data)     { }
6    Integer( const Integer& x ) : m_data(x.m_data) { }
7    ~Integer()                                      { }
8
9    Integer* clone() const
10   {
11     return new Integer( *this );
12   }
13
14   std::string to_str() const
15   {
16     std::stringstream ss;
17     ss << m_data;
18     return ss.str();
19   }
20
21   bool eq( const IObject& rhs ) const
22   {
23     const Integer* rhs_p = dynamic_cast<const Integer*>( &rhs );
24     if ( rhs_p == nullptr )
25       return false;
26     else
27       return ( m_data == rhs_p->m_data );
28   }
29
30   bool lt( const IObject& rhs ) const
31   {
32     const Integer* rhs_p = dynamic_cast<const Integer*>( &rhs );
33     if ( rhs_p == nullptr )
34       return false;
35     else
36       return ( m_data < rhs_p->m_data );
37   }
38
39  private:
40   int m_data;
41 };
```

```
1 int main( void )
2 {
3   Integer a(2);
4   Integer b(3);
5   bool c = ( a == b );
6   return 0;
7 }
```



2.2. Basic List Interface and Implementation

- Object-oriented list which stores `ints`

```
1  class List
2  {
3   public:
4    List();                                  // constructor
5    ~List();                                 // destructor
6    void push_front( int v );                // member function
7
8    struct Node                              //
9    {                                        // nested
10     int   value;                           // struct
11     Node* next_p;                          // definition
12   };                                       //
13
14   Node* m_head_p;                          // member field
15 };
```

- Object-oriented list which stores `IObject*` pointers

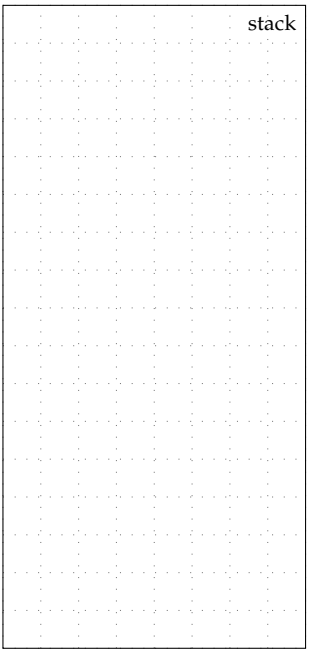
```
1  class List
2  {
3   public:
4    List();                                  // constructor
5    ~List();                                 // destructor
6    void push_front( const IObject& v );     // member function
7
8    struct Node                              //
9    {                                        // nested
10     IObject* obj_p;                        // struct
11     Node*    next_p;                       // definition
12   };                                       //
13
14   Node* m_head_p;                          // member field
15 };
```

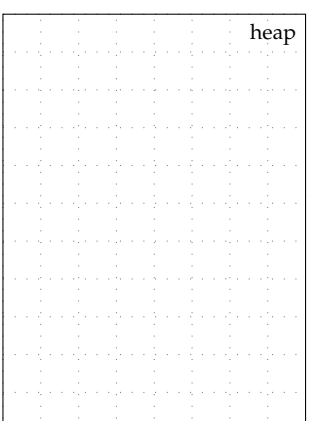
- Implementation of member functions

```
1  List::List()
2   : m_head_p(nullptr)
3  { }
4
5  List::~List()
6  {
7    Node* curr_node_p = m_head_p;
8    while( curr_node_p != nullptr ) {
9      Node* next_node_p = curr_node_p->next_p;
10     delete curr_node_p->obj_p;                // notice!
11     delete curr_node_p;
12     curr_node_p = next_node_p;
13   }
14   m_head_p = nullptr;
15 }
16
17 void List::push_front( const IObject& v )
18 {
19   Node* new_node_p   = new Node();
20   new_node_p->obj_p  = v.clone();
21   new_node_p->next_p = m_head_p;
22   m_head_p           = new_node_p;
23 }
```

- Rule of threes means we also need to declare and define a copy constructor and an overloaded assignment operator

```
1  int main( void )
2  {
3    List list;
4    list.push_front( Integer(12) );
5    list.push_front( Integer(11) );
6    list.push_front( Integer(10) );
7
8    Node* node_p = list.m_head_p;
9    while ( node_p != nullptr ) {
10     int value = node_p->value
11     printf( "%d\n", value );
12     node_p = node_p->next_p;
13   }
14
15   return 0;
16 }
```





2.3. Iterator-Based List Interface and Implementation

- We can use iterators to improve data encapsulation yet still enable the user to cleanly iterate through a sequence

```
1  class List
2  {
3   public:
4
5    class Itr
6    {
7     public:
8      Itr( Node* node_p );
9      void      next();
10     IObject*& get();
11     bool      eq( Itr itr ) const;
12
13    private:
14     friend class List;
15     Node* m_node_p;
16   };
17
18   Itr begin();
19   Itr end();
20   ...
21
22  private:
23
24   struct Node
25   {
26     IObject* obj_p;
27     Node*    next_p;
28   };
29
30   Node* m_head_p;
31
32 };
```

```
List::Itr::Itr( Node* node_p )
 : m_node_p(node_p)
{ }

void List::Itr::next()
{
  assert( m_node_p != nullptr );
  m_node_p = m_node_p->next_p;
}

IObject*& List::Itr::get()
{
  assert( m_node_p != nullptr );
  return m_node_p->obj_p;
}

bool List::Itr::eq( Itr itr ) const
{
  return ( m_node_p == itr.m_node_p );
}

List::Itr List::begin() { return Itr(m_head_p); }
List::Itr List::end()   { return Itr(nullptr);  }
```

- Same overloaded operators as before

```
int main( void )
{
  List list;
  list.push_front( Integer(12) );
  list.push_front( Integer(11) );
  list.push_front( Integer(10) );

  for ( IObject* obj_p : list )
    std::cout << obj_p->to_str() << std::endl;

  return 0;
}
```

```
int main( void )
{
  List list;
  list.push_front( Double(12.3) );
  list.push_front( Double(11.2) );
  list.push_front( Double(10.1) );

  for ( IObject* obj_p : list )
    std::cout << obj_p->to_str() << std::endl;

  return 0;
}
```

```
int main( void )
{
  List list;
  list.push_front( Integer ( 12         ) );
  list.push_front( Double  ( 11.2       ) );
  list.push_front( Complex ( 10.1, 13.5 ) );

  for ( IObject* obj_p : list )
    std::cout << obj_p->to_str() << std::endl;

  return 0;
}
```

2.4. Sorting with Iterators

```
1  void List::sort()
2  {
3    for ( auto itr0 = begin(); itr0 != end(); ++itr0 ) {
4      int max = *itr0;
5      for ( auto itr1 = begin(); itr1 != itr0; ++itr1 ) {
6        if ( max < *itr1 ) {
7          std::swap( max, *itr1 );
8        }
9      }
10     *itr0 = max;
11   }
12 }
13
14 int main( void )
15 {
16   List list0;
17   list0.push_front( Integer(12) );
18   list0.push_front( Integer(11) );
19   list0.push_front( Integer(99) );
20   list0.sort();
21
22   for ( IObject* obj_p : list0 )
23     std::cout << obj_p->to_str() << std::endl;
24
25   List list1;
26   list1.push_front( Double(12.5) );
27   list1.push_front( Double(11.5) );
28   list1.push_front( Double(99.5) );
29   list1.sort();
30
31   for ( IObject* obj_p : list1 )
32     std::cout << obj_p->to_str() << std::endl;
33
34   return 0;
35 }
```

3. Drawing Framework Using Dynamic Polymorphism

```
class Group
{
  ...

  void add( const Point& point )
  {
    assert( m_points_size < 16 );
    m_points[m_points_size] = point; m_points_size++;
  }

  void add( const Line& line )
  {
    assert( m_lines_size < 16 );
    m_lines[m_lines_size] = line; m_lines_size++;
  }

  ...

  void translate( double x_offset, double y_offset )
  {
    for ( int i = 0; i < m_points_size; i++ )
      m_points[i].translate( x_offset, y_offset );
    for ( int i = 0; i < m_lines_size; i++ )
      m_lines[i].translate( x_offset, y_offset );
    for ( int i = 0; i < m_triangles_size; i++ )
      m_triangles[i].translate( x_offset, y_offset );
  }

  ...

 private:
  Point    m_points[16];     int m_points_size;
  Line     m_lines[16];      int m_lines_size;
  Triangle m_triangles[16];  int m_triangles_size;
};
```

```
class Group
{
  ...

  void add( const IShape& shape )
  {
    m_shapes.push_back( shape );
  }

  ...

  void translate( double x_offset, double y_offset )
  {
    for ( auto itr = m_shapes.begin();
          itr != m_shapes.end(); itr++ )
    {
      IShape* shape_p = dynamic_cast<Shape*>(*itr);
      shape->translate( x_offset, y_offset );
    }
  }

 private:
  List m_shapes;
};
```