

ECE 2400 Computer Systems Programming

Fall 2017

Topic 14: C++ Inheritance

School of Electrical and Computer Engineering
Cornell University

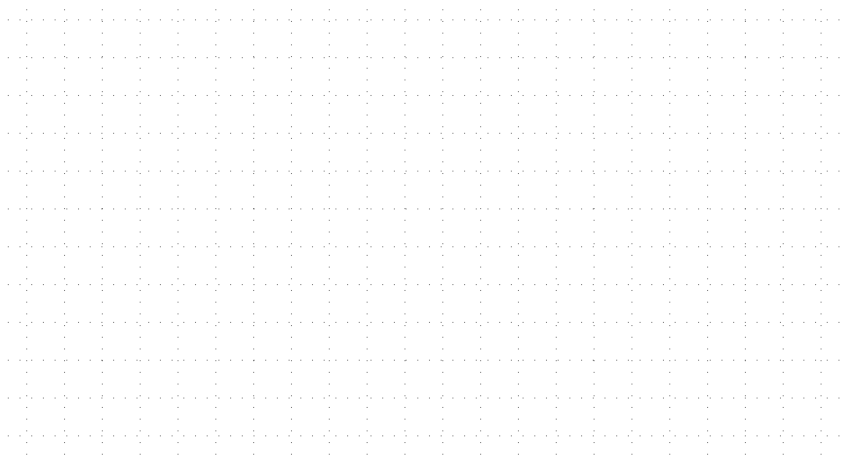
revision: 2017-11-06-09-36

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

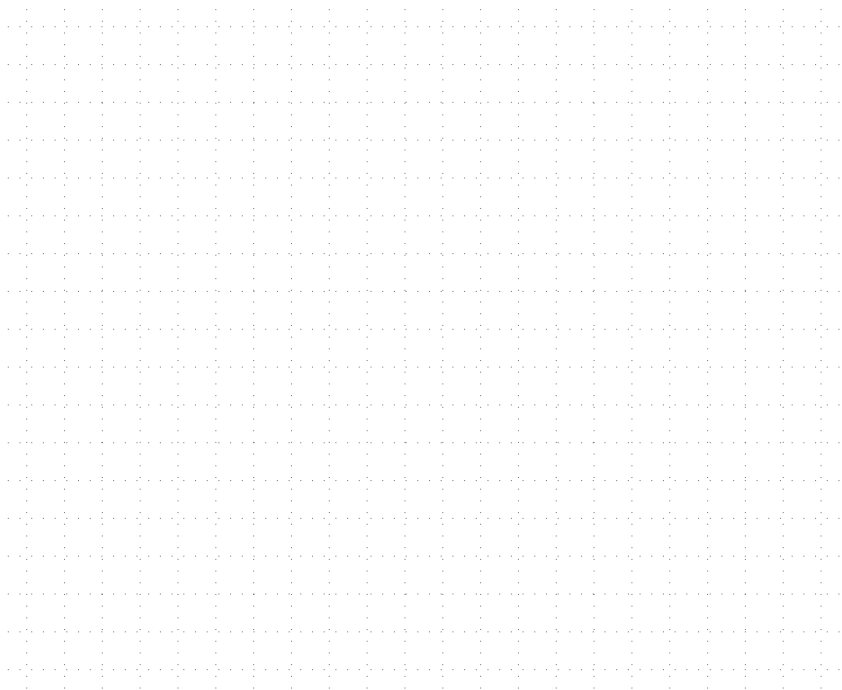
1.1. Object-Oriented Design Without Inheritance



```
1  class Pawn
2  {
3  public:
4
5      Pawn( char col, int row )
6          : m_col(col), m_row(row)
7      { }
8
9      char get_col() const
10     {
11         return m_col;
12     }
13
14     int get_row() const
15     {
16         return m_row;
17     }
18
19     void move( char col, int row )
20     {
21         if ( (col != m_col)
22             || (row != (m_row + 1)) )
23             throw std::invalid_argument();
24
25         m_col = col;
26         m_row = row;
27     }
28
29     std::string to_str() const
30     {
31         std::stringstream ss;
32         ss << "pawn@" << m_col << m_row;
33         return ss.str();
34     }
35
36 private:
37     char m_col;
38     int m_row;
39
40 };
```

```
1  class Rook
2  {
3  public:
4
5      Rook( char col, int row )
6          : m_col(col), m_row(row)
7      { }
8
9      char get_col() const
10     {
11         return m_col;
12     }
13
14     int get_row() const
15     {
16         return m_row;
17     }
18
19     void move( char col, int row )
20     {
21         if ( (col != m_col)
22             && (row != m_row) )
23             throw std::invalid_argument();
24
25         m_col = col;
26         m_row = row;
27     }
28
29     std::string to_str() const
30     {
31         std::stringstream ss;
32         ss << "pawn@" << m_col << m_row;
33         return ss.str();
34     }
35
36 private:
37     char m_col;
38     int m_row;
39
40 };
```

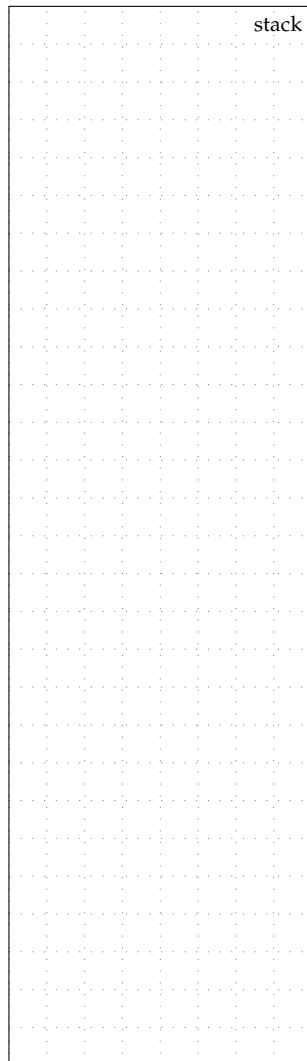
1.2. Inheriting Member Fields and Functions



```
1  class Piece
2  {
3      public:
4
5      Piece( char col, int row )
6          : m_col(col), m_row(row)
7      { }
8
9      char get_col() const
10     {
11         return m_col;
12     }
13
14     int get_row() const
15     {
16         return m_row;
17     }
18
19     // derived classes cannot
20     // access private data in
21     // base class
22     protected:
23
24     char m_col;
25     int  m_row;
26
27 };
```

```
1  class Pawn : public Piece
2  {
3      public:
4
5      Pawn( char col, int row )
6          : Piece( col, row )
7      { }
8
9      void move( char col, int row )
10     {
11         if ( (col != m_col)
12             || (row != (m_row + 1)) )
13             throw std::invalid_argument();
14
15         m_col = col;
16         m_row = row;
17     }
18
19     std::string to_str() const
20     {
21         std::stringstream ss;
22         ss << "pawn@" << m_col << m_row;
23         return ss.str();
24     }
25
26 };
```

```
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  }
```



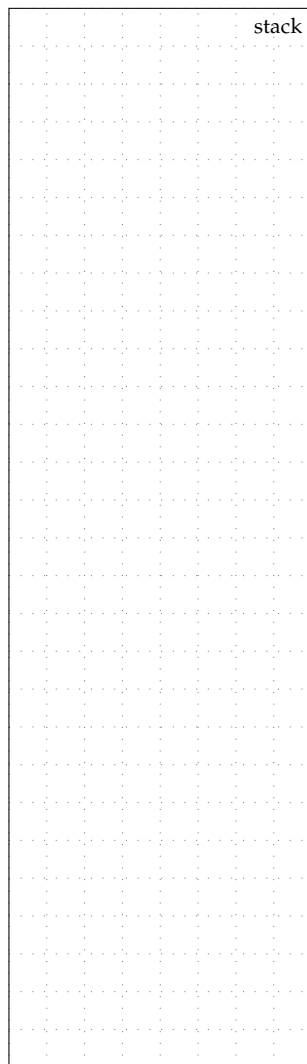
1.3. Method Overriding

```
1  class Piece
2  {
3  public:
4
5      Piece( char col, int row )
6          : m_col(col), m_row(row)
7      { }
8
9      char get_col() const
10     {
11         return m_col;
12     }
13
14     int get_row() const
15     {
16         return m_row;
17     }
18
19     void move( char col, int row )
20     {
21         m_col = col;
22         m_row = row;
23     }
24
25     std::string to_str() const
26     {
27         std::stringstream ss;
28         ss << m_col << m_row;
29         return ss.str();
30     }
31
32 protected:
33
34     char m_col;
35     int m_row;
36
37 };
```

```
1  class Pawn : public Piece
2  {
3  public:
4
5      Pawn( char col, int row )
6          : Piece( col, row )
7      { }
8
9      void move( char col, int row )
10     {
11         if ( (col != m_col)
12             || (row != (m_row + 1)) )
13             throw std::invalid_argument();
14
15         Piece::move( col, row );
16     }
17
18     std::string to_str() const
19     {
20         std::stringstream ss;
21         ss << "pawn@"
22             << Piece::to_str( col, row );
23         return ss.str();
24     }
25
26 };
```

```
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`

```
1  enum PieceType { PAWN, ROOK };
2
3  class Piece
4  {
5  public:
6
7      Piece( PieceType type, char col, int row )
8          : m_type(type), m_col(col), m_row(row)
9      { }
10
11     PieceType get_type() const
12     {
13         return m_type;
14     }
15
16     ...
17
18     protected:
19     PieceType m_type;
20     char      m_col;
21     int       m_row;
22 };
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
```

```
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 }
```

stack

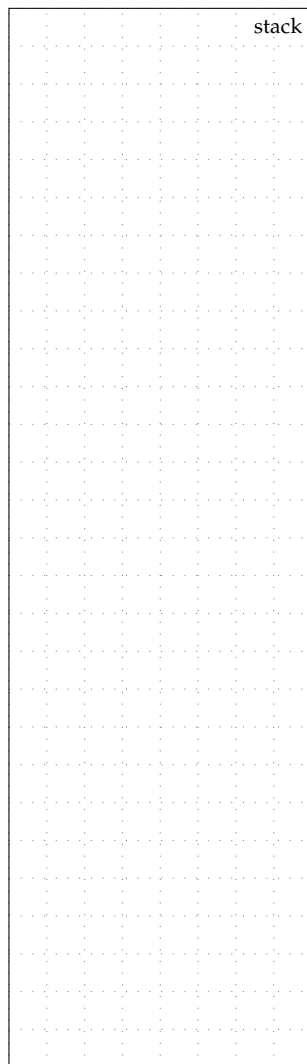
1.4. Dynamic Polymorphism via Virtual Methods

```
1  class Piece
2  {
3  public:
4
5      Piece( char col, int row )
6          : m_col(col), m_row(row)
7      { }
8
9      char get_col() const
10     {
11         return m_col;
12     }
13
14     int get_row() const
15     {
16         return m_row;
17     }
18
19     virtual
20     void move( char col, int row )
21     {
22         m_col = col;
23         m_row = row;
24     }
25
26     virtual
27     std::string to_str() const
28     {
29         std::stringstream ss;
30         ss << m_col << m_row;
31         return ss.str();
32     }
33
34     protected:
35
36     char m_col;
37     int m_row;
38
39 };
```

```
1  class Pawn : public Piece
2  {
3  public:
4
5      Pawn( char col, int row )
6          : m_col(col), m_row(row)
7      { }
8
9      void move( char col, int row )
10     {
11         if ( (col != m_col)
12             || (row != (m_row + 1)) )
13             throw std::invalid_argument();
14
15         Piece::move( col, row );
16     }
17
18     std::string to_str() const
19     {
20         std::stringstream ss;
21         ss << "pawn@"
22             << Piece::to_str( col, row );
23         return ss.str();
24     }
25
26 };
```

```
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 polymorphism** 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 }
```

stack

1.5. Abstract Base Classes

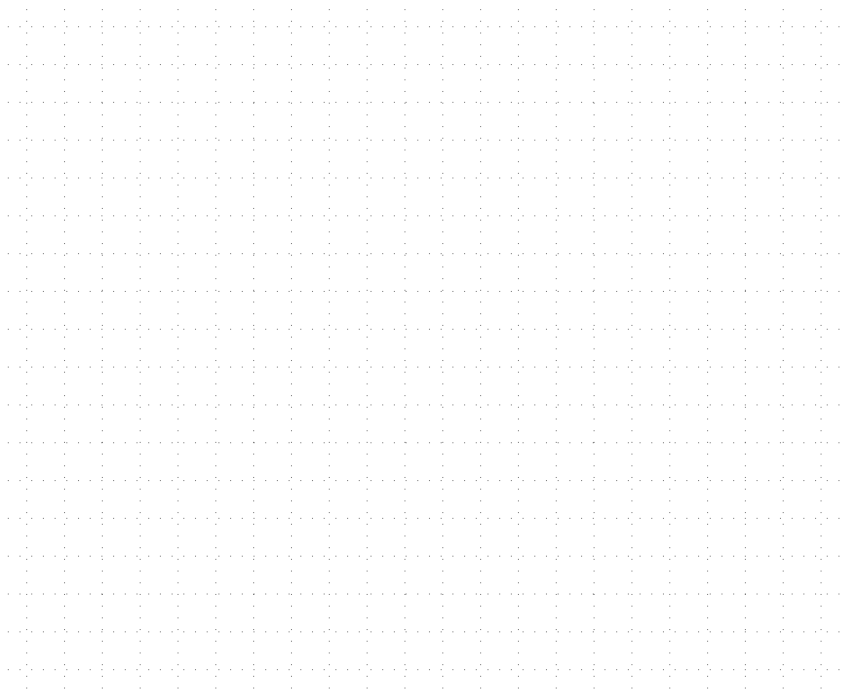
```
1  class IPiece
2  {
3  public:
4
5      virtual
6      ~IPiece() { };
7
8      virtual
9      char get_col() const = 0;
10
11     virtual
12     int get_row() const = 0;
13
14     virtual
15     void move( char col, int row ) = 0;
16
17     virtual
18     std::string to_str() const = 0;
19
20 };

1  class Pawn : public IPiece
2  {
3  public:
4
5      Pawn( char col, int row )
6          : m_col(col), m_row(row)
7      { }
8
9      ~Pawn()
10     { }
11
12     char get_col() const
13     {
14         return m_col;
15     }
16
17     int get_row() const
18     {
19         return m_row;
20     }
21
22     void move( char col, int row )
23     {
24         if ( (col != m_col)
25             || (row != (m_row + 1)) )
26             throw std::invalid_argument();
27
28         m_col = col;
29         m_row = row;
30     }
31
32     std::string to_str() const
33     {
34         std::stringstream ss;
35         ss << "pawn@" << m_col << m_row;
36         return ss.str();
37     }
38
39 private:
40     char m_col;
41     int m_row;
42 };
```

```
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 }
```

stack

1.6. Revisiting Composition vs. Generalization



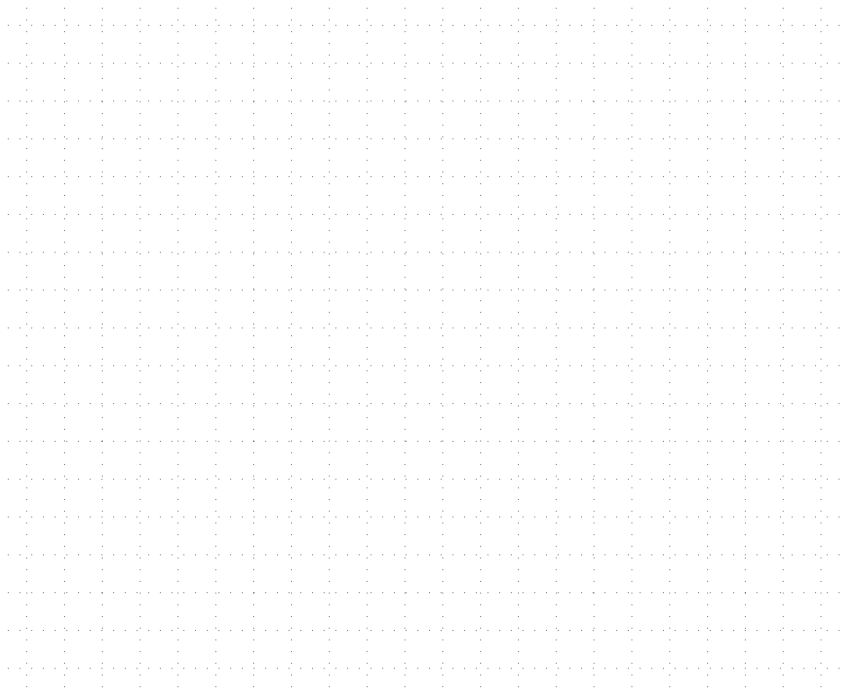
```
1 class Position
2 {
3 public:
4     Position( char col, int row )
5         : m_col(col), m_row(row)
6     { }
7
8     char get_col() const
9     {
10         return m_col;
11     }
12
13     int get_row() const
14     {
15         return m_row;
16     }
17
18     void move( char col, int row )
19     {
20         m_col = col;
21         m_row = row;
22     }
23
24     std::string to_str() const
25     {
26         std::stringstream ss;
27         ss << m_col << m_row;
28         return ss.str();
29     }
30
31 private:
32     char m_col;
33     int m_row;
34 };
```

```
1 class Pawn : public IPiece
2 {
3 public:
4     Pawn( char col, int row )
5         : m_pos(col,row)
6     { }
7
8     ~Pawn()
9     { }
10
11     char get_col() const
12     {
13         return m_pos.get_col();
14     }
15
16     int get_row() const
17     {
18         return m_pos.get_row();
19     }
20
21     void move( char col, int row )
22     {
23         int curr_col = m_pos.get_col();
24         int curr_row = m_pos.get_row();
25         if ( col != curr_col
26             || (row != (curr_row + 1)) )
27             throw std::invalid_argument();
28
29         m_pos.move( col, row );
30     }
31
32     std::string to_str() const
33     {
34         std::stringstream ss;
35         ss << "pawn@" << m_pos.to_str();
36         return ss.str();
37     }
38
39 private:
40     Position m_pos;
41 };
```

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

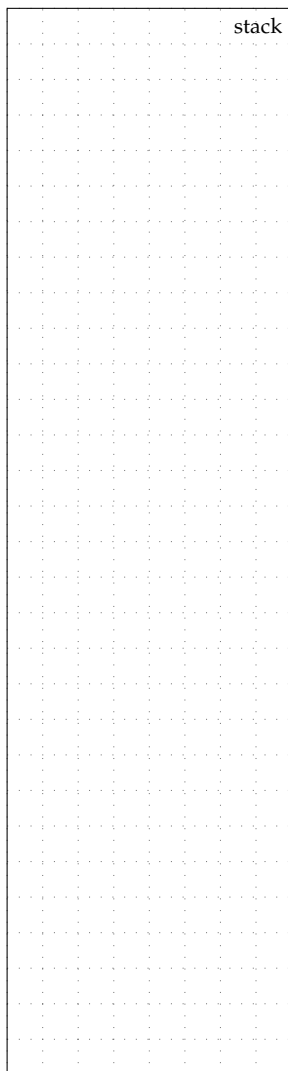
2.1. Class Hierarchy for Objects



```
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

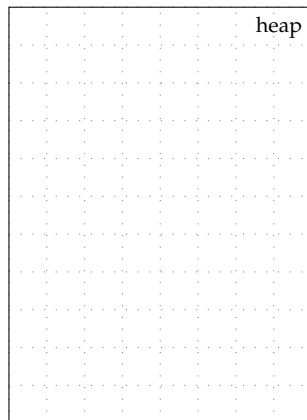
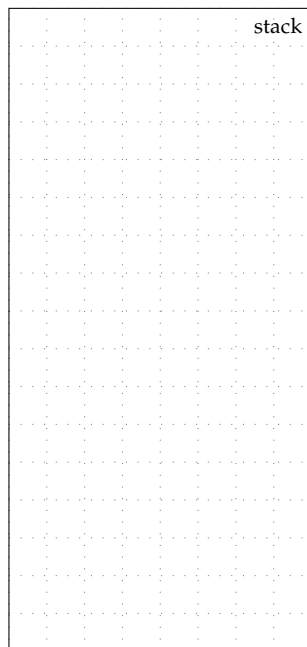
```
1  class List
2  {
3  public:
4      List();                // constructor
5      ~List();              // destructor
6      void push_front( const IObject& v ); // member function
7
8      struct Node            //
9      {                    // nested
10         IObject* obj_p;     // struct
11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

```
1  List::Itr::Itr( Node* node_p )
2      : m_node_p(node_p)
3  { }
4
5  void List::Itr::next()
6  {
7      assert( m_node_p != nullptr );
8      m_node_p = m_node_p->next_p;
9  }
10
11  IObject*& List::Itr::get()
12  {
13      assert( m_node_p != nullptr );
14      return m_node_p->obj_p;
15  }
16
17  bool List::Itr::eq( Itr itr ) const
18  {
19      return ( m_node_p == itr.m_node_p );
20  }
21
22  List::Itr List::begin() { return Itr(m_head_p); }
23  List::Itr List::end()  { return Itr(nullptr); }
```

- Same overloaded operators as before

```
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      for ( IObject* obj_p : list )
9          std::cout << obj_p->to_str() << std::endl;
10
11     return 0;
12 }

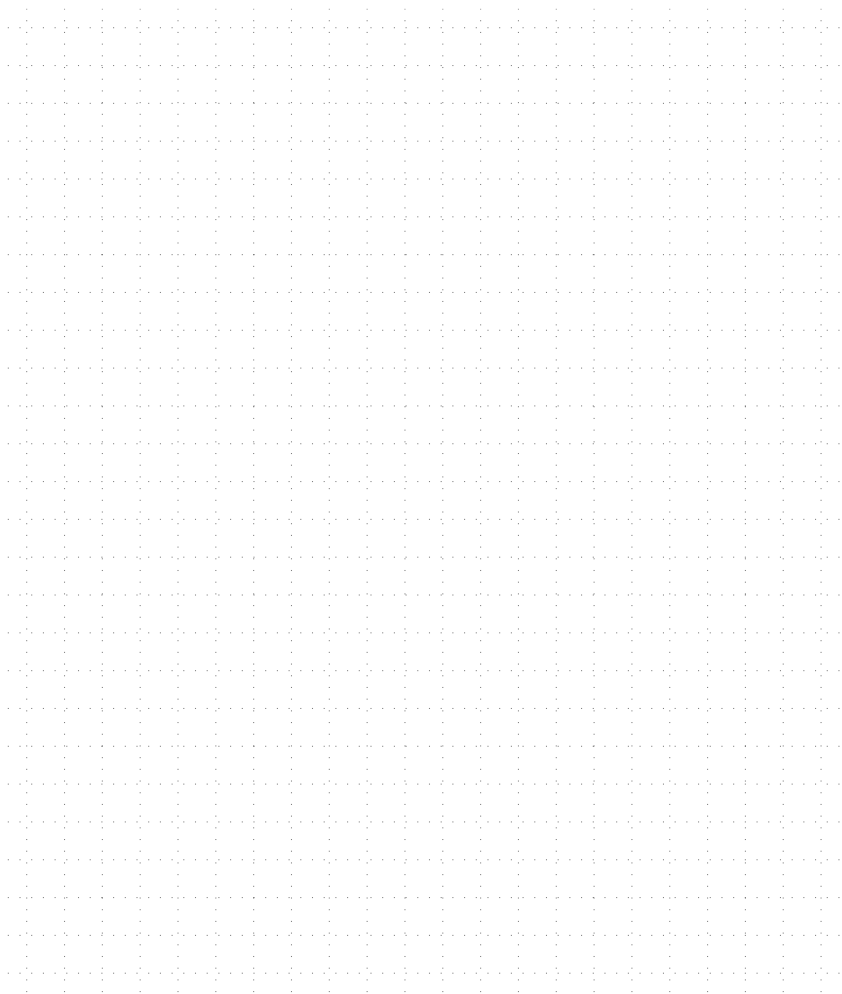
1  int main( void )
2  {
3      List list;
4      list.push_front( Double(12.3) );
5      list.push_front( Double(11.2) );
6      list.push_front( Double(10.1) );
7
8      for ( IObject* obj_p : list )
9          std::cout << obj_p->to_str() << std::endl;
10
11     return 0;
12 }

1  int main( void )
2  {
3      List list;
4      list.push_front( Integer ( 12          ) );
5      list.push_front( Double  ( 11.2        ) );
6      list.push_front( Complex ( 10.1, 13.5 ) );
7
8      for ( IObject* obj_p : list )
9          std::cout << obj_p->to_str() << std::endl;
10
11     return 0;
12 }
```

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



```
1  class Group
2  {
3      ...
4
5      void add( const Point& point )
6      {
7          assert( m_points_size < 16 );
8          m_points[m_points_size] = point; m_points_size++;
9      }
10
11     void add( const Line& line )
12     {
13         assert( m_lines_size < 16 );
14         m_lines[m_lines_size] = line; m_lines_size++;
15     }
16
17     ...
18
19     void translate( double x_offset, double y_offset )
20     {
21         for ( int i = 0; i < m_points_size; i++ )
22             m_points[i].translate( x_offset, y_offset );
23         for ( int i = 0; i < m_lines_size; i++ )
24             m_lines[i].translate( x_offset, y_offset );
25         for ( int i = 0; i < m_triangles_size; i++ )
26             m_triangles[i].translate( x_offset, y_offset );
27     }
28
29     ...
30
31     private:
32         Point    m_points[16];        int m_points_size;
33         Line     m_lines[16];         int m_lines_size;
34         Triangle m_triangles[16];     int m_triangles_size;
35     };
```

```
1  class Group
2  {
3      ...
4
5      void add( const IShape& shape )
6      {
7          m_shapes.push_back( shape );
8      }
9
10     ...
11
12     void translate( double x_offset, double y_offset )
13     {
14         for ( auto itr = m_shapes.begin();
15              itr != m_shapes.end(); itr++ )
16         {
17             IShape* shape_p = dynamic_cast<Shape*>(*itr);
18             shape->translate( x_offset, y_offset );
19         }
20     }
21
22     private:
23         List m_shapes;
24 };
```