

ECE 2400 Computer Systems Programming

Spring 2025

Topic 12: Object-Oriented Programming

School of Electrical and Computer Engineering
Cornell University

revision: 2025-03-19-12-18

1	C++ Classes	5
1.1.	C++ Member Functions	7
1.2.	C++ Constructors	11
1.3.	C++ Operator Overloading	13
1.4.	C++ Rule of Three	16
1.5.	C++ Scope-Bound Resource Management	25
1.6.	C++ Data Encapsulation	26
2	Object-Oriented Data Structures	28
2.1.	Singly Linked List Interface	28
2.2.	Singly Linked List Implementation	29
2.3.	Iterator-Based List Interface and Implementation	32
3	C++ Inheritance	36
3.1.	C++ Implementation Inheritance	38
3.2.	From Implementation to Interface Inheritance	42
3.3.	C++ Interface Inheritance	49

3.4. Revisiting Composition vs. Generalization	52
4 OO Data Structures w/Dynamic Polymorphism	54
4.1. Singly Linked List Interface	58
4.2. Singly Linked List Implementation	59
5 Drawing Framework Case Study	61

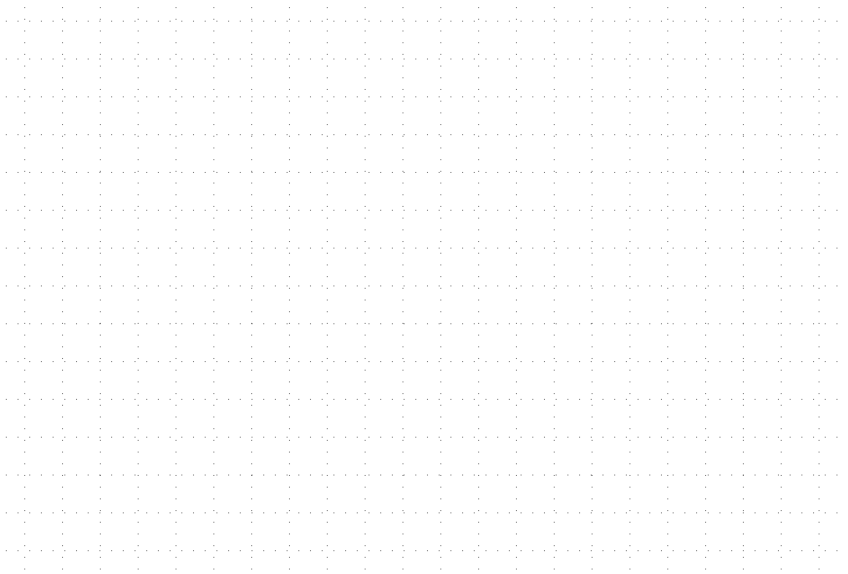
zyBooks logo indicates readings and coding labs in the course zyBook which will not be discussed in detail in lecture. Students are responsible for all material covered in lecture and in the course zyBook.

Copyright © 2025 Anne Bracy. All rights reserved. This handout was prepared by Prof. Anne Bracy at Cornell University for ECE 2400 / ENGRD 2140 Computer Systems Programming (derived from previous handouts prepared and copyrighted by Prof. Christopher Batten). Download and use of this handout is permitted for individual educational non-commercial purposes only. Redistribution either in part or in whole via both commercial or non-commercial means requires written permission.

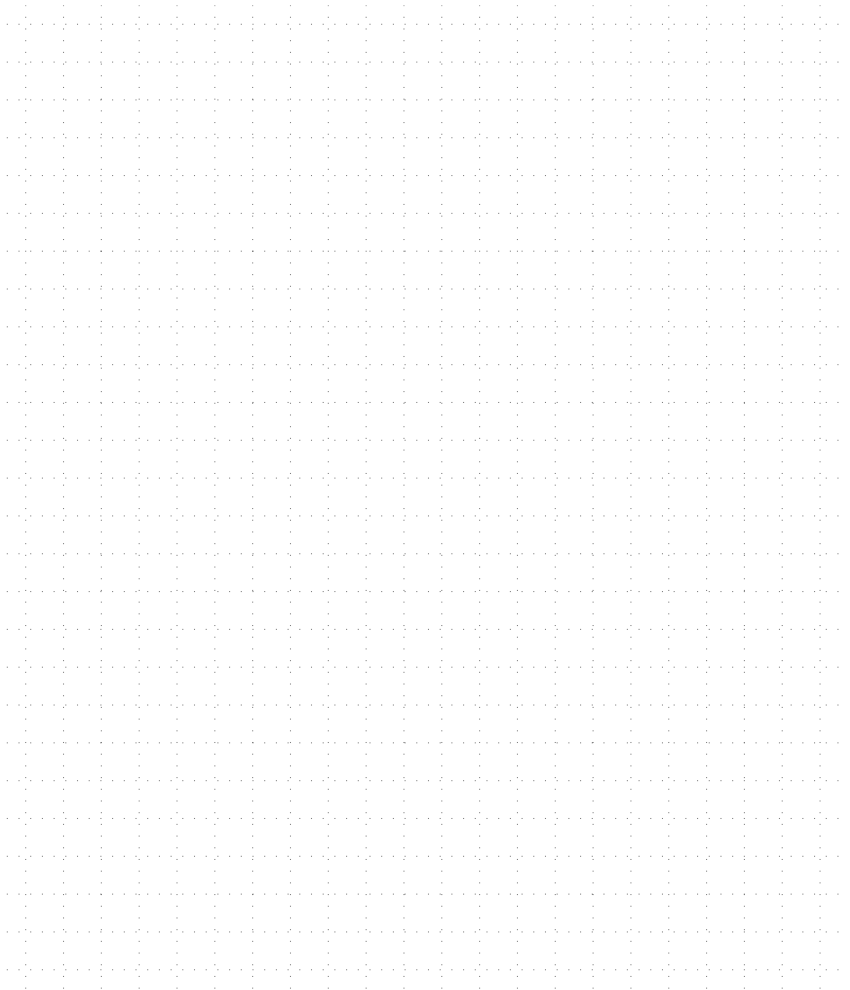
Object-oriented programming

- Programming is organized around defining, instantiating, and manipulating *objects* which contain data (i.e., fields, attributes) and code (i.e., methods)
- Classes are the “types” of objects, objects are instances of classes
- **Classes** are nouns, **methods** are verbs/actions
- Classes are organized according to various relationships
 - **composition** relationship (“Class X has a Y”)
 - **generalization** relationship (“Class X is a Y”)
 - **association** relationship (“Class X acts on Y”)

Example class diagram for animals



Example class diagram for shapes and drawings

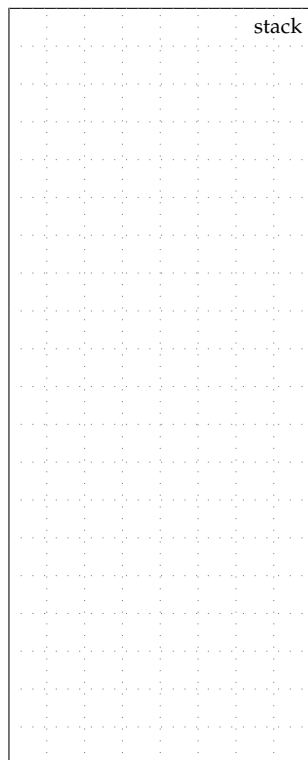


1. C++ Classes

- Perfectly possible to use object-oriented programming in C

```
1  typedef struct
2  {
3      double x;
4      double y;
5  }
6  point_t;
7
8  void point_translate( point_t* this,
9                      double x_offset, double y_offset )
10 {
11     this->x += x_offset; this->y += y_offset;
12 }
13
14 void point_scale( point_t* this, double factor )
15 {
16     this->x *= factor; this->y *= factor;
17 }
18
19 void point_rotate( point_t* this, double angle )
20 {
21     const double pi = 3.14159265358979323846;
22     double s = std::sin((angle*pi)/180);
23     double c = std::cos((angle*pi)/180);
24
25     double x_new = (c * this->x) - (s * this->y);
26     double y_new = (s * this->x) + (c * this->y);
27
28     this->x = x_new; this->y = y_new;
29 }
```

```
01 int main( void )
02 {
03     point_t pt;
04     pt.x = 0;
05     pt.y = 0;
06
07     point_translate( &pt, 1, 2 );
08     point_scale     ( &pt, 2 );
09     return 0;
10 }
```



1.1. C++ Member Functions

- C++ allows functions to be defined *within* the struct namespace
- C++ struct has both **member fields** and **member functions**
- Member functions have an implicit `this` pointer
- Member functions which do not modify fields are `const`

```
1  struct Point
2  {
3      double x; // member fields
4      double y; //
5
6      // member functions
7
8      void point_translate( double x_offset, double y_offset )
9      {
10         this->x += x_offset; this->y += y_offset;
11     }
12
13     void point_scale( double factor )
14     {
15         this->x *= factor; this->y *= factor;
16     }
17
18     void point_rotate( double angle )
19     {
20         ...
21         double x_new = (c * this->x) - (s * this->y);
22         double y_new = (s * this->x) + (c * this->y);
23         this->x = x_new; this->y = y_new;
24     }
25 };
```

- Non-static member functions are accessed using the dot (.) operator in the same way we access fields

```

01 int main( void )
02 {
03     Point pt;
04     pt.x = 0;
05     pt.y = 0;
06
07     pt.point_translate( 1, 2 );
08     pt.point_scale( 2 );
09     return 0;
10 }

```

- Recall object-oriented prog in C

```

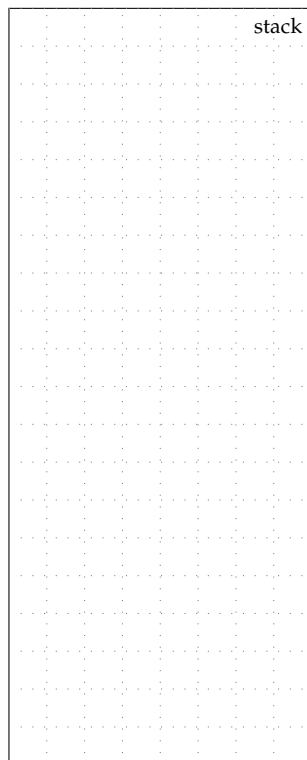
1  int main( void )
2  {
3      point_t pt;
4      pt.x = 0;
5      pt.y = 0;
6
7      point_translate( &pt, 1, 2 );
8      point_scale    ( &pt, 2 );
9      return 0;
10 }

```

```

point_translate( &pt, 1, 2 ) ↔ pt.point_translate( 1, 2 )
foo( &bar, ... ) ↔ bar.foo( ... )

```

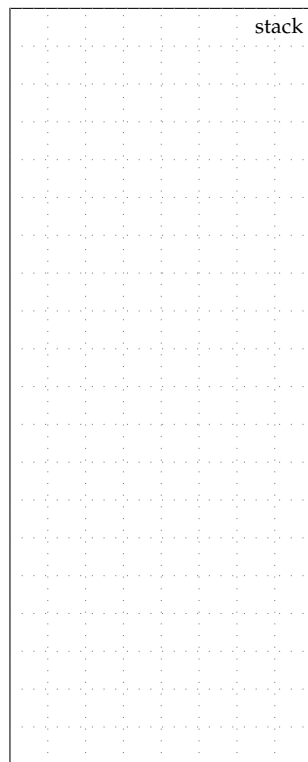


- Member functions are in struct namespace
- No need to use the `point_` prefix
- Member fields are in scope within every member function
- No need to explicitly use `this` pointer

```
1  struct Point
2  {
3      double x; // member fields
4      double y; //
5
6      // member functions
7
8      void translate( double x_offset, double y_offset )
9      {
10         x += x_offset; y += y_offset;
11     }
12
13     void scale( double factor )
14     {
15         x *= factor; y *= factor;
16     }
17
18     void rotate( double angle )
19     {
20         ...
21         double x_new = (c * x) - (s * y);
22         double y_new = (s * x) + (c * y);
23         x = x_new; y = y_new;
24     }
25 };
```

Draw a state diagram corresponding to the execution of this program

```
□□□ 01 int main( void )
□□□ 02 {
□□□ 03     Point pt;
□□□ 04     pt.x = 0;
□□□ 05     pt.y = 0;
□□□ 06
□□□ 07     pt.translate( 1, 2 );
□□□ 08     pt.scale( 2 );
□□□ 09     return 0;
□□□ 10 }
```



- A class is just a struct with member functions
- An object is just an instance of a struct with member functions

1.2. C++ Constructors

- We want to avoid the user from directly accessing member fields
- We want to ensure an object is always initialized to a known state
- In C, we used `foo_construct`
- In C++, we could add a `construct` member function

```
1  int main( void )
2  {
3      Point pt;
4      pt.construct();
5      pt.translate( 1, 2 );
6      pt.scale( 2 );
7      return 0;
8  }
```

- What if we call `translate` before `construct`?
- What if we call `construct` multiple times?
- We want a way to specify a special “constructor” member function
 - *always* called when you create an object
 - cannot be called directly, can *only* be called during object creation
- C++ adds support for language-level constructors (i.e., special member functions)
 - no return type
 - same name as the class
- Can use function overloading to have many different constructors

Draw a state diagram corresponding to the execution of this program

```
01 struct Point
02 {
03     double x;
04     double y;
05
06     // default constructor
07     Point()
08     {
09         x = 0;
10         y = 0;
11     }
12
13     // non-default constructor
14     Point( double x_, double y_ )
15     {
16         x = x_;
17         y = y_;
18     }
19
20     void translate( double x_offset,
21                   double y_offset )
22     {
23         x += x_offset; y += y_offset;
24     }
25
26 };
27
28 int main( void )
29 {
30     Point pt0;
31     Point pt1( 1, 2 );
32     pt0 = pt1;
33     pt0.translate( 1, 2 );
34     return 0;
35 }
```

stack

- Constructors automatically called with `new`

```
1 Point* pt0_p = new Point;           // constructor called
2 Point* pt1_p = new Point[4];        // constructor called 4 times
```

- Initialization lists initialize members before body of constructor
 - Avoids creating a temporary default object
 - Required for initializing reference members

1	<code>struct Point</code>	1	<code>struct Point</code>
2	<code>{</code>	2	<code>{</code>
3	<code>double x;</code>	3	<code>double x;</code>
4	<code>double y;</code>	4	<code>double y;</code>
5		5	
6	<code>// default constructor</code>	6	<code>// default constructor</code>
7	<code>Point()</code>	7	<code>Point()</code>
8	<code>{ x = 0; y = 0; }</code>	8	<code>: x(0), y(0) { }</code>
9		9	
10	<code>// non-default constructor</code>	10	<code>// non-default constructor</code>
11	<code>Point(double x_, double y_)</code>	11	<code>Point(double x_, double y_)</code>
12	<code>{ x = x_; y = y_; }</code>	12	<code>: x(x_), y(y_) { }</code>
13		13	
14	<code>...</code>	14	<code>...</code>
15	<code>};</code>	15	<code>};</code>

1.3. C++ Operator Overloading

- C++ **operator overloading** enables using built-in operators (e.g., `+`, `-`, `*`, `/`) with user-defined types
- Applying an operator to a user-defined type essentially calls a function (either a member function or an overloaded free function)

```
01 Point operator+( Point pt0,  
02                   Point pt1 )  
03 {  
04     pt0.translate( pt1.x, pt1.y );  
05     return pt0;  
06 }  
07  
08 int main( void )  
09 {  
10     Point ptA(1,2);  
11     Point ptB(3,4);  
12     Point ptC;  
13     ptC = ptA + ptB;  
14     return 0;  
15 }
```

stack

stack

```
1 Point operator+( const Point& pt0, const Point& pt1 )
2 {
3     Point tmp = pt0;
4     tmp.translate( pt1.x, pt1.y );
5     return tmp;
6 }
7
8 Point operator*( const Point& pt, double factor )
9 {
10    Point tmp = pt;
11    tmp.scale( factor );
12    return tmp;
13 }
14
15 Point operator*( double factor, const Point& pt )
16 {
17    Point tmp = pt;
18    tmp.scale( factor );
19    return tmp;
20 }
21
22 Point operator%( const Point& pt, double angle )
23 {
24    Point tmp = pt;
25    tmp.rotate( angle );
26    return tmp;
27 }
```

- Operator overloading enables elegant syntax for user-defined types

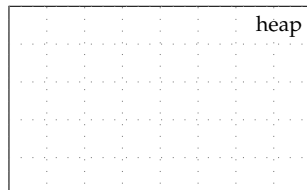
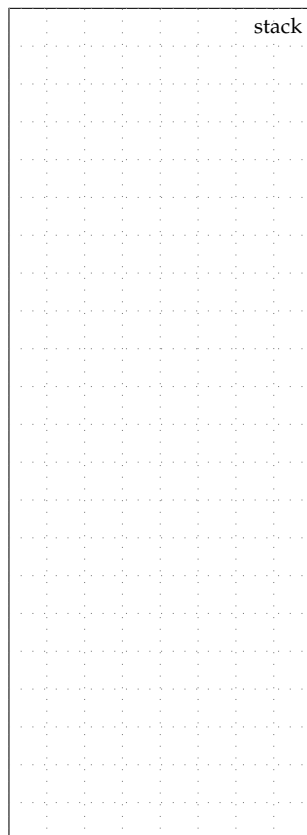
```
1 Point pt0(1,2);
2 pt0.translate(5,3);
3 pt0.rotate(45);
4 pt0.scale(1.5);
5 Point pt1 = pt0;

1 Point pt0(1,2);
2 Point pt1 = 1.5 * ( ( pt0 + Point(5,3) ) % 45 );
```

1.4. C++ Rule of Three

- What if point coordinates are allocated on the heap?

```
01 struct DPoint
02 {
03     double* x_p;
04     double* y_p;
05
06     DPoint() {
07         x_p = new double;
08         y_p = new double;
09         *x_p = 0;
10         *y_p = 0;
11     }
12
13     void translate( double x_offset,
14                   double y_offset )
15     {
16         *x_p += x_offset;
17         *y_p += y_offset;
18     }
19     ...
20 };
21
22 int main( void )
23 {
24     DPoint pt0;
25     pt0.translate( 1, 2 );
26     return 0;
27 }
```



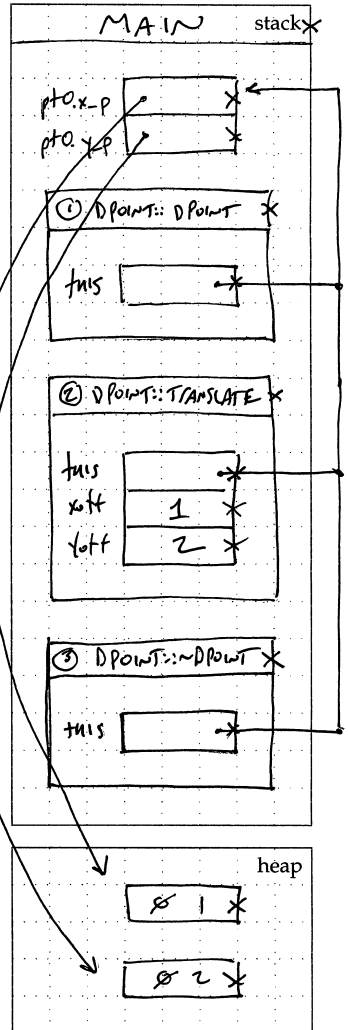
C++ Destructors

- Special member function to destroy an object

```

000001 struct DPoint
000002 {
000003     double* x_p;
000004     double* y_p;
000005
000006     DPoint() {
000007         x_p = new double;
000008         y_p = new double;
000009         *x_p = 0;
000010         *y_p = 0;
000011     }
000012
000013     ~DPoint() {
000014         delete x_p;
000015         delete y_p;
000016     }
000017
000018     void translate( double x_offset,
000019                   double y_offset )
000020     {
000021         *x_p += x_offset;
000022         *y_p += y_offset;
000023     }
000024
000025     ...
000026 };
000027
000028 int main( void )
000029 {
000030     DPoint pt0;
000031     pt0.translate( 1, 2 );
000032     return 0;
000033 }

```

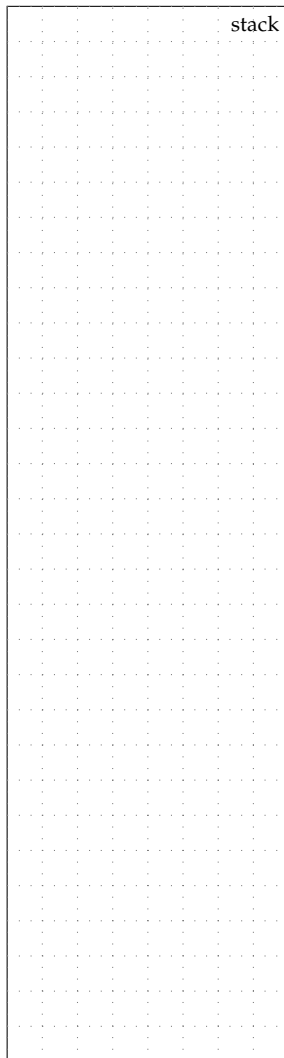
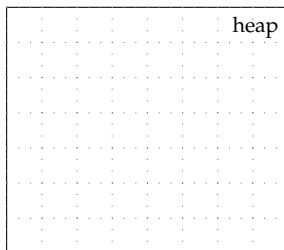


- What if we copy an object with dynamically allocated memory?

```

0000 01 struct DPoint
0000 02 {
0000 03     double* x_p;
0000 04     double* y_p;
0000 05
0000 06     DPoint() {
0000 07         x_p = new double;
0000 08         y_p = new double;
0000 09         *x_p = 0;
0000 10         *y_p = 0;
0000 11     }
0000 12
0000 13     ~DPoint() {
0000 14         delete x_p; delete y_p;
0000 15     }
0000 16     ...
0000 17 };
0000 18
0000 19 int main( void )
0000 20 {
0000 21     DPoint pt0;
0000 22     DPoint pt1 = pt0;
0000 23     pt0.translate( 1, 2 );
0000 24     return 0;
0000 25 }

```



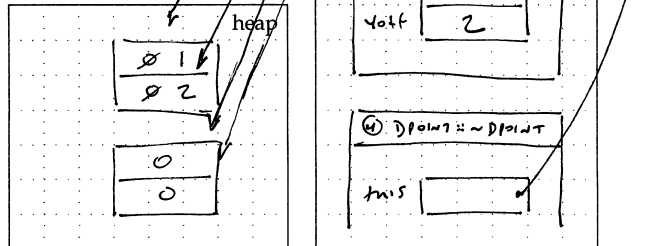
C++ Copy Constructors

- Special member function to construct a new object from an old object

```

000001 struct DPoint
000002 {
000003     double* x_p;
000004     double* y_p;
000005
000006     DPoint( const DPoint& pt ) {
000007         x_p = new double;
000008         y_p = new double;
000009         *x_p = *pt.x_p;
000010         *y_p = *pt.y_p;
000011     }
000012
000013     ~DPoint() {
000014         delete x_p; delete y_p;
000015     }
000016     ...
000017 };
000018
000019 int main( void )
000020 {
000021     DPoint pt0;
000022     DPoint pt1 = pt0;
000023     pt0.translate( 1, 2 );
000024     return 0;
000025 }

```

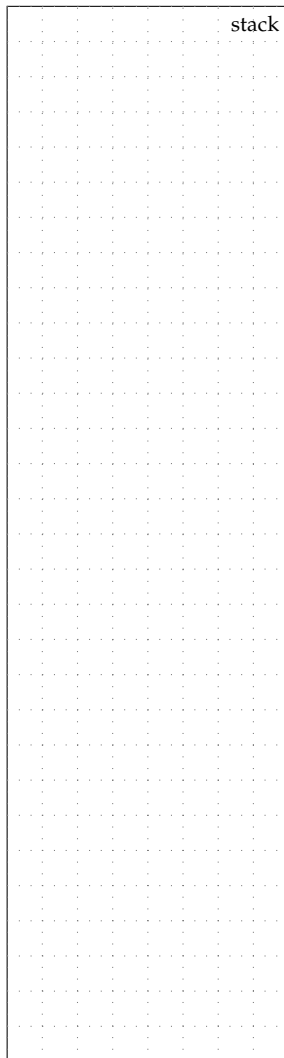
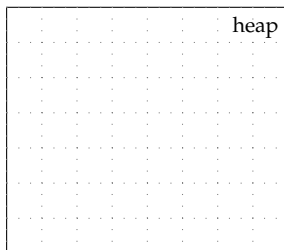


- What if we assign to an object with dynamically allocated memory?

```

01 struct DPoint
02 {
03     double* x_p;
04     double* y_p;
05
06     DPoint() {
07         x_p = new double;
08         y_p = new double;
09         *x_p = 0;
10         *y_p = 0;
11     }
12
13     ~DPoint() {
14         delete x_p; delete y_p;
15     }
16     ...
17 };
18
19 int main( void )
20 {
21     DPoint pt0;
22     DPoint pt1;
23     pt1 = pt0;
24     pt0.translate( 1, 2 );
25     return 0;
26 }

```



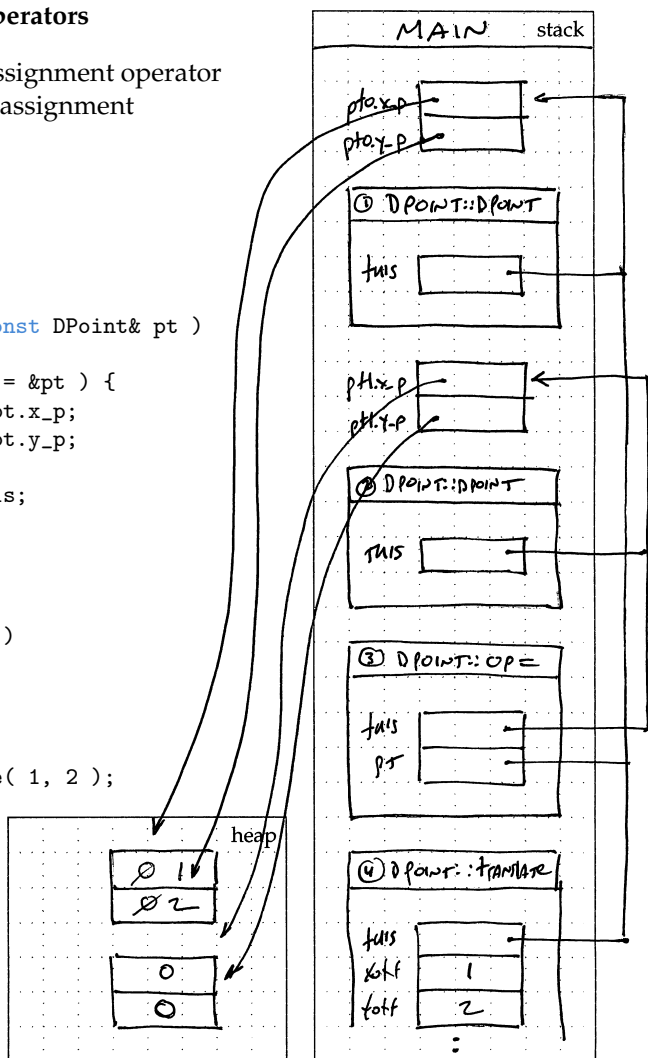
C++ Assignment Operators

- An overloaded assignment operator will be called for assignment

```

01 struct DPoint
02 {
03     double* x_p;
04     double* y_p;
05
06     DPoint&
07     operator=( const DPoint& pt )
08     {
09         if ( this != &pt ) {
10             *x_p = *pt.x_p;
11             *y_p = *pt.y_p;
12         }
13         return *this;
14     }
15     ...
16 };
17
18 int main( void )
19 {
20     DPoint pt0;
21     DPoint pt1;
22     pt1 = pt0;
23     pt0.translate( 1, 2 );
24     return 0;
25 }

```



C++ Rule of Three

- Default **destructor**, **copy constructor**, and **assignment operator** will work fine for simple classes
- For a more complex class may need to define one of these ...
- ... and if you define one, then you probably need to define all three!
- Be very careful about self assignment

```

1  struct DPoint
2  {
3      double* x_p;
4      double* y_p;
5
6      DPoint()
7      {
8          x_p = new double;
9          y_p = new double;
10         *x_p = 0;
11         *y_p = 0;
12     }
13
14     DPoint( double x, double y )
15     {
16         x_p = new double;
17         y_p = new double;
18         *x_p = x;
19         *y_p = y;
20     }

```

```

1      DPoint( const DPoint& pt )
2      {
3          x_p = new double;
4          y_p = new double;
5          *x_p = *pt.x_p;
6          *y_p = *pt.y_p;
7      }
8
9      ~DPoint()
10     {
11         delete x_p;
12         delete y_p;
13     }
14
15     DPoint&
16     operator=( const DPoint& pt )
17     {
18         if ( this != &pt ) {
19             *x_p = *pt.x_p;
20             *y_p = *pt.y_p;
21         }
22         return *this;
23     }
24
25     ...
26 };

```

Label all calls to the rule of three member functions

- Only label *non-trivial* destruction, initialization, assignment
- D = destructor, CC = copy constructor, AO = assignment operator

```

1  void ex0()
2  {
3      Point pt0(1,2);
4      Point pt1(3,4);
5      pt0 = p2;
6  }
7
8  void ex1()
9  {
10     DPoint pt0(1,2);
11     DPoint pt1( pt0 );
12     DPoint pt2 = pt1;
13     pt0 = p2;
14     pt0 = p2 = p1;
15     pt0 = pt0;
16 }
17
18 DPoint foo( DPoint pt )
19 {
20     return pt;
21 }
22
23 void ex2()
24 {
25     DPoint pt0(1,2);
26     DPoint pt1 = foo( pt0 );
27     DPoint pt2;
28     pt2 = foo( pt1 );
29 }
30
31 void ex3()
32 {
33     DPoint pt0(1,2);
34     DPoint pt1(3,4);
35     DPoint* a = &pt0;
36     DPoint* b = a;
37 }
38
39 void ex4()
40 {
41     DPoint* pt0 = new DPoint(1,2);
42 }
43
44 void ex5()
45 {
46     DPoint* pt0 = new DPoint(1,2);
47     delete pt0;
48 }
49
50 void ex6()
51 {
52     DPoint* pt0 = new DPoint[4];
53     delete[] pt0;
54 }
55
56 struct TwoPoints
57 {
58     DPoint pt0;
59     DPoint pt1;
60 }
61
62 void ex7()
63 {
64     TwoPoints pts0;
65     TwoPoints pts1( pts0 );
66     TwoPoints pts2 = pts1;
67     pts0 = pts2;
68 }
69
70 void ex8()
71 {
72     DPoint pt0;
73     throw -1;
74     DPoint pt1;
75 }

```

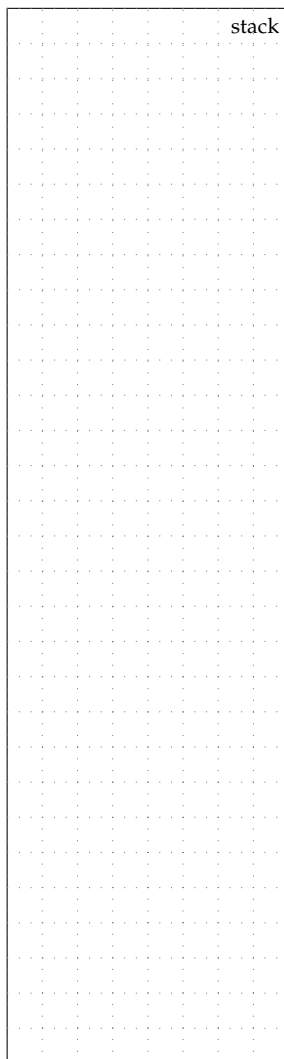
C++ Exceptions and Destructors

- Destructors called automatically for all objects in scope when exception thrown

```

01 struct DPoint
02 {
03     double* x_p;
04     double* y_p;
05
06     ~DPoint() {
07         delete x_p;
08         delete y_p;
09     }
10
11     void translate( double x_offset,
12                   double y_offset )
13     {
14         if ( (x_offset > 100)
15             || (y_offset > 100) )
16             throw 42;
17         *x_p += x_offset;
18         *y_p += y_offset;
19     }
20
21     ...
22 };
23
24 int main( void )
25 {
26     try {
27         DPoint pt0;
28         pt0.translate( 1e9, 0 );
29     }
30     catch ( int e ) {
31         return e;
32     }
33     return 0;
34 }

```



1.5. C++ Scope-Bound Resource Management

- **Scope-bound resource management** is a design pattern that ties a resource to object lifetime (**RAII: resource acquisition is initialization**)
- Use `new` in constructors and `delete` in destructors
- Elegantly ensures `delete` is called for every `new` even if an exception is thrown; can completely eliminate memory leaks

```
1  DPoint transform( DPoint pt0, DPoint pt1, double scale )
2  {
3      if ( scale < 0 )
4          throw -1;
5
6      DPoint pt;
7
8      if ( scale == 0 ) {
9          pt = DPoint(0,0);
10     }
11     else {
12         DPoint pt2 = pt0 + pt1;
13         DPoint pt3 = pt2 * scale;
14         pt = pt3;
15     }
16
17     return pt;
18 }
19
20 int main( void )
21 {
22     DPoint pt0(1,2);
23     DPoint pt1(3,4);
24     DPoint pt3 =
25         transform( pt0, pt1, 2.0 );
26     return 0;
27 }
```

- `new` and `delete` do not appear anywhere in this code!
- **Scope-bound resource management** completely takes care of all dynamic memory allocation and ensures there are no memory leaks
- While our `DPoint` example is admittedly contrived, scope-bound resource management is absolutely critical to the C++ implementation of:
 - strings
 - data structures
 - file I/O
 - smart pointers
 - threads
 - locks

1.6. C++ Data Encapsulation

- Recall the importance of separating **interface** from **implementation**
- This is an example of **abstraction**
- In this context, also called **information hiding**, **data encapsulation**
 - Hides implementation complexity
 - Can change implementation without impacting users
- So far, we have relied on a *policy* to enforce data encapsulation
 - Users of a struct could still directly access member fields

```
1  int main( void )
2  {
3      Point pt(1,2);
4      pt.x = 13; // direct access to member fields
5      return 0;
6  }
```

- In C++, we can *enforce* data encapsulation at compile time
 - By default all member fields and functions of a struct are **public**
 - Member fields and functions can be explicitly labeled as **public** or **private**
 - Externally accessing an internal private field causes a compile time error

```
1  struct Point
2  {
3      private:
4          double m_x; double m_y;
5
6      public:
7          // default constructor
8          Point() { m_x = 0; m_y = 0; }
9
10         // non-default constructor
11         Point( double x, double y ) { m_x = x; m_y = y; }
12         ...
13     };
```

- In C++, we usually use `class` instead of `struct`
 - By default all member fields and functions of a `struct` are `public`
 - By default all member fields and functions of a `class` are `private`
 - We should almost always use `class` and explicitly use `public` and `private`

```
1  class Point // almost always use class instead of struct
2  {
3      public:    // always explicitly use public ...
4      private:  // ... or private
5  };
```

- We are free to change how we store the point
- We could change point to store coordinates on the stack or heap
- Statically guaranteed that others cannot access this private implementation

2. Object-Oriented Data Structures

- Object-oriented programming can enable elegant interfaces and implementations for data structures

2.1. Singly Linked List Interface

- Recall the interface for a C singly linked list data structure

```
1  typedef struct
2  {
3      // implementation specific
4  }
5  slist_int_t;
6
7  void slist_int_construct ( slist_int_t* this );
8  void slist_int_destruct ( slist_int_t* this );
9  void slist_int_push_front ( slist_int_t* this, int v );
10 ...
```

- Corresponding interface for a C++ singly linked list data structure

```
1  class SListInt
2  {
3      public:
4          SListInt();           // constructor
5          ~SListInt();          // destructor
6          void push_front( int v ); // member function
7          ...
8
9          // implementation specific
10 };
```

- C-based list could not be easily copied or assigned
- C++ rule of three means we also need to declare and define a copy constructor and an overloaded assignment operator

2.2. Singly Linked List Implementation

- Recall the implementation for a C singly linked list data structure

```
1 typedef struct _slist_int_node_t
2 {
3     int value;
4     struct _slist_int_node_t* next_p;
5 }
6 slist_int_node_t;
7
8 typedef struct
9 {
10     slist_int_node_t* head_p;
11 }
12 list_int_t;
```

- Corresponding implementation for a C++ singly linked list data structure

```
1 class SListInt
2 {
3     public:
4         SListInt();           // constructor
5         ~SListInt();          // destructor
6         void push_front( int v ); // member function
7         ...
8
9         struct Node           // nested struct declaration
10        {
11            int value;
12            Node* next_p;
13        };
14
15        Node* m_head_p;        // member field
16    };
```

- Implementation for a C singly linked list data structure

```

1  void slist_int_construct(
2      slist_int_t* this )
3  {
4      this->head_p = NULL;
5  }
6
7  void slist_int_push_front(
8      slist_int_t* this, int v )
9  {
10     slist_int_node_t* new_node_p
11         = malloc(sizeof(slist_int_node_t));
12     new_node_p->value = v;
13     new_node_p->next_p = this->head_p;
14     this->head_p = new_node_p;
15 }
16
17 void slist_int_destruct(
18     slist_int_t* this )
19 {
20     while ( this->head_p != NULL ) {
21         list_int_node_t* temp_p
22             = this->head_p->next_p;
23         free( this->head_p );
24         this->head_p = temp_p;
25     }
26 }
```

- Implementation for a C++ singly linked list data structure

```

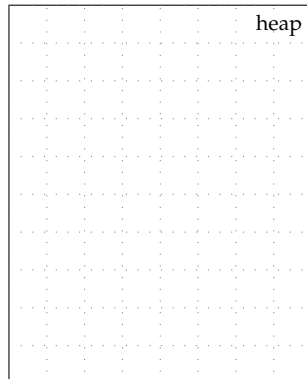
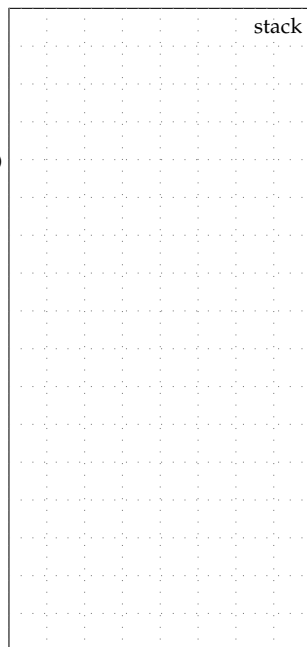
1  SListInt::SListInt()
2
3  {
4      m_head_p = nullptr;
5  }
6
7  void SListInt::push_front( int v )
8
9  {
10     Node* new_node_p
11         = new Node;
12     new_node_p->value = v;
13     new_node_p->next_p = m_head_p;
14     m_head_p = new_node_p;
15 }
16
17 SListInt::~~SListInt()
18
19 {
20     while ( m_head_p != nullptr ) {
21         Node* temp_p
22             = m_head_p->next_p;
23         delete m_head_p;
24         m_head_p = temp_p;
25     }
26 }
```

- Notice the syntax used for separating member function *declarations* from member function *definitions*

```

01 SListInt::SListInt()
02 {
03     m_head_p = nullptr;
04 }
05
06 void SListInt::push_front( int v )
07 {
08     Node* new_node_p
09         = new Node;
10     new_node_p->value = v;
11     new_node_p->next_p = m_head_p;
12     m_head_p = new_node_p;
13 }
14
15 int main( void )
16 {
17     SListInt lst;
18     lst.push_front(12);
19     lst.push_front(11);
20     lst.push_front(10);
21
22     SListInt::Node* curr_p
23         = lst.m_head_p;
24     while ( curr_p != nullptr ) {
25         int value = curr_p->value;
26         curr_p = curr_p->next_p;
27     }
28
29     return 0;
30 }

```



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

```

1  SListInt::Itr::Itr( Node* node_p )
2      { m_node_p = node_p; }
3
4  void SListInt::Itr::next()
5  {
6      assert( m_node_p != nullptr );
7      m_node_p = m_node_p->next_p;
8  }
9
10 int& SListInt::Itr::get()
11 {
12     assert( m_node_p != nullptr );
13     return m_node_p->value;
14 }
15
16 bool SListInt::Itr::eq( Itr itr ) const
17 {
18     return ( m_node_p == itr.m_node_p );
19 }
20
21 SListInt::Itr SListInt::begin()
22 {
23     return Itr(m_head_p);
24 }
25
26 SListInt::Itr SListInt::end()
27 {
28     return Itr(nullptr);
29 }

```



```

1  SListInt::Node* curr_p          1  SListInt::Itr itr
2      = list.m_head_p;           2      = list.begin();
3  while ( curr_p != nullptr ) {  3  while ( !itr.eq(list.end()) ) {
4      int value = curr_p->value    4      int value = itr.get();
5      printf( "%d\n", value );    5      printf( "%d\n", value );
6      curr_p = curr_p->next_p;    6      itr.next();
7  }                               7  }

```

- We can use operator overloading to improve iterator syntax

```

1  // postfix increment operator (itr++)
2  SListInt::Itr operator++( SListInt::Itr& itr, int )
3  {
4      SListInt::Itr itr_tmp = itr; itr.next(); return itr_tmp;
5  }
6
7  // prefix increment operator (++itr)
8  SListInt::Itr& operator++( SListInt::Itr& itr )
9  {
10     itr.next(); return itr;
11 }
12
13 // dereference operator (*itr)
14 int& operator*( SListInt::Itr& itr )
15 {
16     return itr.get();
17 }
18
19 // not-equal operator (itr0 != itr1)
20 bool operator!=( const SListInt::Itr& itr0,
21                 const SListInt::Itr& itr1 )
22 {
23     return !itr0.eq( itr1 );
24 }

```



```

1  SListInt::Itr itr = lst.begin();    1  SListInt::Itr itr = lst.begin();
2  while ( !itr.eq(lst.end()) ) {      2  while ( itr != lst.end() ) {
3      int value = itr.get();           3      int value = *itr;
4      printf( "%d\n", value );         4      printf( "%d\n", value );
5      itr.next();                     5      ++itr;
6  }                                   6  }

```



```

1  for ( SListInt::Itr itr = lst.begin(); itr != lst.end(); ++itr ) {
2      printf( "%d\n", *itr );
3  }

```

- We can use auto and range-based loops with user-defined data structures to enable elegant syntax
- C++11 auto keyword will automatically infer type from initializer

```
1  for ( auto itr = lst.begin(); itr != lst.end(); ++itr ) {  
2      printf( "%d\n", *itr );  
3  }
```

- C++11 range-based loops are syntactic sugar for above

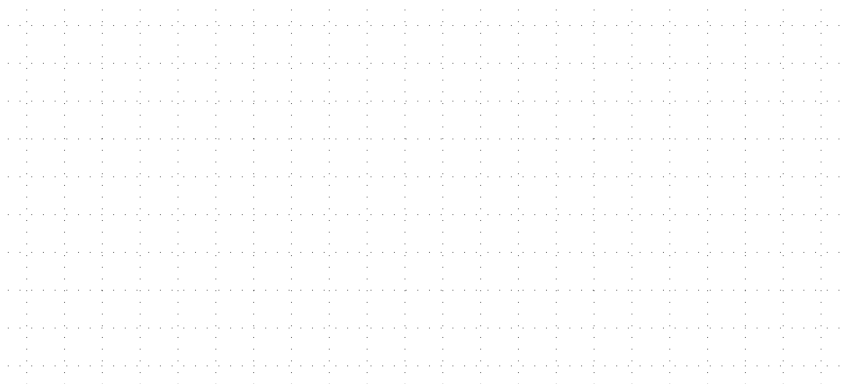
```
1  for ( int v : lst ) {  
2      printf( "%d\n", v );  
3  }
```

3. C++ Inheritance

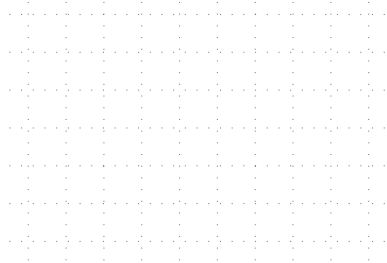
```
1  class Pawn
2  {
3  public:
4
5      Pawn( char col, int row )
6      {
7          m_col = col;
8          m_row = row;
9      }
10
11     int get_pts() const
12     {
13         return 1;
14     }
15
16     char get_col() const
17     {
18         return m_col;
19     }
20
21     int get_row()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

```
1  class Rook
2  {
3  public:
4
5      Rook( char col, int row )
6      {
7          m_col = col;
8          m_row = row;
9      }
10
11     int get_pts() const
12     {
13         return 5;
14     }
15
16     char get_col() const
17     {
18         return m_col;
19     }
20
21     int get_row() const
22     {
23         return m_row;
24     }
25
26     void move( char col, int row )
27     {
28         if ( (col != m_col)
29             && (row != m_row) )
30             throw 42;
31
32         m_col = col;
33         m_row = row;
34     }
35
36 private:
37     char m_col;
38     int m_row;
39 };
```

- Object-oriented design without inheritance

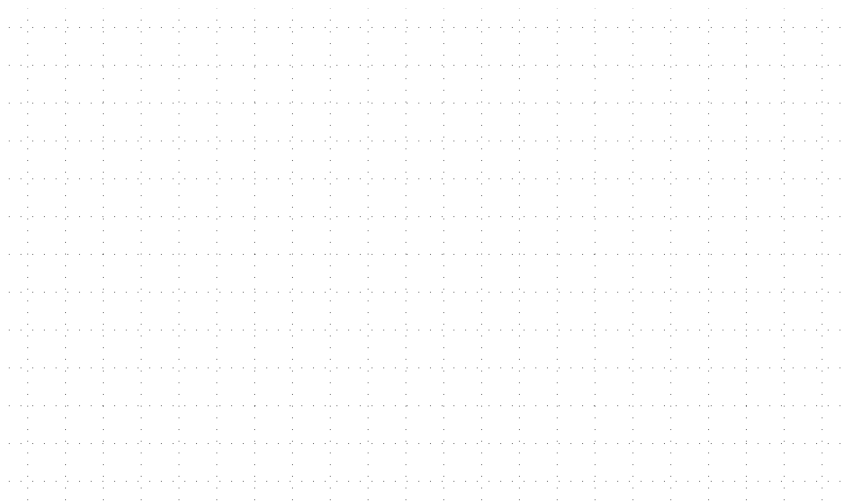


- We want to be able to ...
 - ... create algorithms and data structures that work for all pieces
 - ... easily reuse implementation code across pieces
- Inheritance enables declaring a **derived** class that automatically includes the interface and implementation of a different **base** class
 - Derived 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

```
1  class Piece
2  {
3  public:
4
5      Piece( char col, int row )
6      {
7          m_col = col;
8          m_row = row;
9      }
10
11     char get_col() const
12     {
13         return m_col;
14     }
15
16     int get_row() const
17     {
18         return m_row;
19     }
20
21
22
23
24
25
26     void move( char col, int row )
27     {
28         m_col = col;
29         m_row = row;
30     }
31
32
33
34
35     // derived classes cannot
36     // access private data in
37     // base class
38     protected:
39         char m_col;
40         int m_row;
41
42 };
```

```
1  class Rook : public Piece
2  {
3  public:
4
5      Rook( char col, int row )
6      : Piece( col, row )
7      { }
8
9
10
11
12
13
14
15
16
17
18
19
20
21     int get_pts() const
22     {
23         return 5;
24     }
25
26     void move( char col, int row )
27     {
28         if ( (col != m_col)
29             && (row != m_row) )
30             throw 42;
31
32         Piece::move( col, row );
33     }
34
35
36
37
38
39
40
41
42 };
```

```
000001 class Piece
000002 {
000003     Piece( char col, int row )
000004     {
000005         m_col = col;
000006         m_row = row;
000007     }
000008
000009     char get_col() const {
000010         return m_col;
000011     }
000012
000013     int get_row() const {
000014         return m_row;
000015     }
000016     ...
000017 };
000018
000019 class Pawn : public Piece
000020 {
000021     Pawn( char col, int row )
000022         : Piece( col, row )
000023     { }
000024
000025     int get_pts() const {
000026         return 1;
000027     }
000028     ...
000029 };
000030
000031 int main( void )
000032 {
000033     Pawn p('a',2);
000034     char col = p.get_col();
000035     return 0;
000036 }
```

stack

```

0001 class Piece
0002 {
0003     Piece( char col, int row ) {
0004         m_col = col;
0005         m_row = row;
0006     }
0007
0008     char get_col() const {
0009         return m_col;
0010     }
0011
0012     int get_row() const {
0013         return m_row;
0014     }
0015     ...
0016 };
0017
0018 class Pawn : public Piece
0019 {
0020     int get_pts() const { return 1; }
0021     ...
0022 };
0023
0024 class Rook : public Piece
0025 {
0026     int get_pts() const { return 5; }
0027     ...
0028 };
0029
0030 int main( void )
0031 {
0032     Pawn p('a',2);
0033     Rook r('h',3);
0034     int pts0 = p.get_pts();
0035     int pts1 = r.get_pts();
0036     int sum = pts0 + pts1;
0037     return 0;
0038 }

```

<i>no c/d</i>	<i>c</i>	<i>d</i>	<i>c/d</i>	<i>stack</i>
1	1	1	1	1
2	2	2	1	2
3	3	3	1	3
4	4	4	1	4
5	5	5	1	5
6	6	6	1	6
7	7	7	1	7
8	8	8	1	8
9	9	9	1	9
10	10	10	1	10
11	11	11	1	11
12	12	12	1	12
13	13	13	1	13
14	14	14	1	14
15	15	15	1	15
16	16	16	1	16
17	17	17	1	17
18	18	18	1	18
19	19	19	1	19
20	20	20	1	20
21	21	21	1	21
22	22	22	1	22
23	23	23	1	23
24	24	24	1	24
25	25	25	1	25
26	26	26	1	26
27	27	27	1	27
28	28	28	1	28
29	29	29	1	29
30	30	30	1	30
31	31	31	1	31
32	32	32	1	32
33	33	33	1	33
34	34	34	1	34
35	35	35	1	35
36	36	36	1	36
37	37	37	1	37
38	38	38	1	38
39	39	39	1	39
40	40	40	1	40
41	41	41	1	41
42	42	42	1	42
43	43	43	1	43
44	44	44	1	44
45	45	45	1	45
46	46	46	1	46
47	47	47	1	47
48	48	48	1	48
49	49	49	1	49
50	50	50	1	50
51	51	51	1	51
52	52	52	1	52
53	53	53	1	53
54	54	54	1	54
55	55	55	1	55
56	56	56	1	56
57	57	57	1	57
58	58	58	1	58
59	59	59	1	59
60	60	60	1	60
61	61	61	1	61
62	62	62	1	62
63	63	63	1	63
64	64	64	1	64
65	65	65	1	65
66	66	66	1	66
67	67	67	1	67
68	68	68	1	68
69	69	69	1	69
70	70	70	1	70
71	71	71	1	71
72	72	72	1	72
73	73	73	1	73
74	74	74	1	74
75	75	75	1	75
76	76	76	1	76
77	77	77	1	77
78	78	78	1	78
79	79	79	1	79
80	80	80	1	80
81	81	81	1	81
82	82	82	1	82
83	83	83	1	83
84	84	84	1	84
85	85	85	1	85
86	86	86	1	86
87	87	87	1	87
88	88	88	1	88
89	89	89	1	89
90	90	90	1	90
91	91	91	1	91
92	92	92	1	92
93	93	93	1	93
94	94	94	1	94
95	95	95	1	95
96	96	96	1	96
97	97	97	1	97
98	98	98	1	98
99	99	99	1	99
100	100	100	1	100

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();
14
15 // explicit type casting to wrong type (segfault)
16 Pawn* pawn_p = (Pawn*) piece_p;
17
18 // ptr == nullptr if piece_p is not pointer to a Rook
19 Rook* ptr = dynamic_cast<Rook*>(piece_p);
```

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  }
```

[illegible]

```

1  // enum for all possible
2  // implementation types
3  enum PieceType { PAWN, ROOK };
4
5  class Piece
6  {
7  public:
8
9      Piece( PieceType type,
10             char col, int row )
11      {
12          m_type = type;
13          m_col = col;
14          m_row = row;
15      }
16
17      PieceType get_type() const
18      {
19          return m_type;
20      }
21
22      char get_col() const
23      {
24          return m_col;
25      }
26
27      int get_row() const
28      {
29          return m_row;
30      }
31
32      void move( char col, int row )
33      {
34          m_col = col;
35          m_row = row;
36      }
37
38  protected:
39
40      PieceType m_type; // explicit type field
41      char      m_col;
42      int       m_row;
43
44  };

```

- Add type field to base class
- Use type field to determine implementation type
- Cast a pointer from the base class to a pointer to the appropriate derived type
- Call the corresponding member function

```

1  class Rook : public Piece
2  {
3  public:
4
5      Rook( char col, int row )
6          : Piece( ROOK, col, row )
7      { }
8
9      int get_pts() const
10     {
11         return 5;
12     }
13
14     void move( char col, int row )
15     {
16         if ( (col != m_col)
17             && (row != m_row) )
18             throw 42;
19
20         Piece::move( col, row );
21     }
22
23 };

```



```

1  class Piece
2  {
3  public:
4
5      Piece( PieceType type,
6             char col, int row )
7      {
8          m_type = type;
9          m_col  = col;
10         m_row  = row;
11     }
12
13     int get_pts() const
14     {
15         if ( m_type == PAWN ) {
16             Pawn* pawn_p = (Pawn*) this;
17             return pawn_p->get_pts();
18         }
19         else if ( m_type == ROOK ) {
20             Rook* rook_p = (Rook*) this;
21             return rook_p->get_pts();
22         }
23     }
24
25     char get_col() const
26     {
27         return m_col;
28     }
29
30     int get_row() const
31     {
32         return m_row;
33     }
34
35     void move( char col, int row )
36     {
37         m_col = col;
38         m_row = row;
39     }
40
41 protected:
42     PieceType m_type;
43     char      m_col;
44     int       m_row;
45 };

```

- Add type field to base class
- Use type field to determine implementation type
- Cast a pointer from the base class to a pointer to the appropriate derived type
- Call the corresponding member function
- Example of **dynamic dispatch**

```

1  class Rook : public Piece
2  {
3  public:
4
5      Rook( char col, int row )
6          : Piece( ROOK, col, row )
7      { }
8
9      int get_pts() const
10     {
11         return 5;
12     }
13
14     void move( char col, int row )
15     {
16         if ( (col != m_col)
17             && (row != m_row) )
18             throw 42;
19
20         Piece::move( col, row );
21     }
22
23 };

```

```

000001 class Piece
000002 {
000003     ...
000004     int get_pts() const
000005     {
000006         if ( m_type == PAWN ) {
000007             Pawn* pawn_p = (Pawn*) this;
000008             return pawn_p->get_pts();
000009         }
000010         else if ( m_type == ROOK ) {
000011             Rook* rook_p = (Rook*) this;
000012             return rook_p->get_pts();
000013         }
000014     }
000015 };
000016
000017 class Pawn : public Piece
000018 {
000019     int get_pts() const { return 1; }
000020     ...
000021 };
000022
000023 int calc_pts( Piece* p0, Piece* p1 )
000024 {
000025     int pts0 = p0->get_pts();
000026     int pts1 = p1->get_pts();
000027     return pts0 + pts1;
000028 }
000029
000030 int main( void )
000031 {
000032     Pawn p('a',2);
000033     Rook r('h',3);
000034     int sum = calc_pts( &p, &r );
000035     return 0;
000036 }

```

no c/d stack

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

```
1  class Piece
2  {
3  public:
4
5      Piece( char col, int row )
6      {
7          m_col = col;
8          m_row = row;
9      }
10
11     virtual int get_pts() const = 0;
12
13     char get_col() const
14     {
15         return m_col;
16     }
17
18     int get_row() const
19     {
20         return m_row;
21     }
22
23     void move( char col, int row )
24     {
25         m_col = col;
26         m_row = row;
27     }
28
29     protected:
30     char m_col;
31     int m_row;
32 };

1  class Rook : public Piece
2  {
3  public:
4
5      Rook( char col, int row )
6          : Piece( col, row )
7      { }
8
9      int get_pts() const
10     {
11         return 5;
12     }
13
14     void move( char col, int row )
15     {
16         if ( (col != m_col)
17             && (row != m_row) )
18             throw 42;
19
20         Piece::move( col, row );
21     }
22
23     };
```

```

0000 01 class Piece
0000 02 {
0000 03     virtual int get_pts() const = 0;
0000 04     ...
0000 05 };
0000 06
0000 07 class Pawn : public Piece
0000 08 {
0000 09     int get_pts() const { return 1; }
0000 10     ...
0000 11 };
0000 12
0000 13 class Rook : public Piece
0000 14 {
0000 15     int get_pts() const { return 5; }
0000 16     ...
0000 17 };
0000 18
0000 19 int calc_pts( Piece* p0, Piece* p1 )
0000 20 {
0000 21     int pts0 = p0->get_pts();
0000 22     int pts1 = p1->get_pts();
0000 23     return pts0 + pts1;
0000 24 }
0000 25
0000 26 int main( void )
0000 27 {
0000 28     Pawn p('a',2);
0000 29     Rook r('h',3);
0000 30     int sum = calc_pts( &p, &r );
0000 31     return 0;
0000 32 }

```

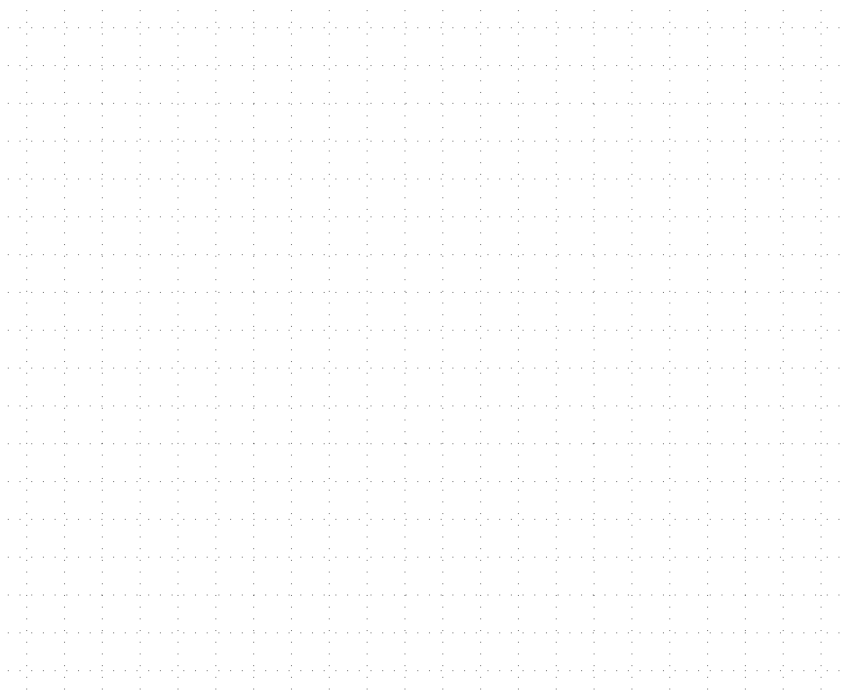
no c/d stack

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



```

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

```

```

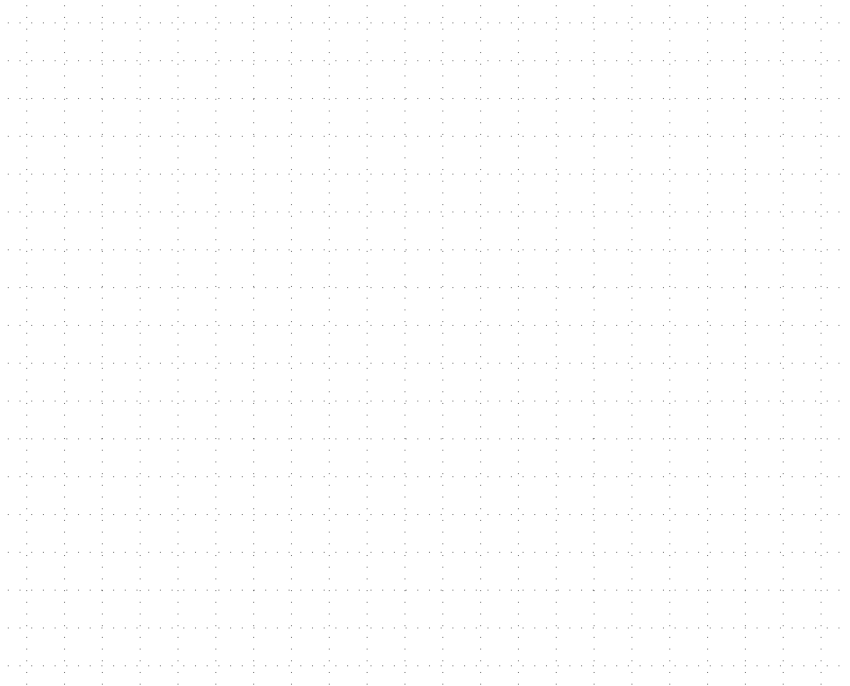
1  class Pawn : public IPiece
2  {
3  public:
4
5      Pawn( char col, int row )
6      {
7          m_pos = Position(col,row);
8      }
9
10     ~Pawn()
11     { }
12
13     int get_pts() const
14     {
15         return 1;
16     }
17
18     char get_col() const
19     {
20         return m_pos.get_col();
21     }
22
23     int get_row() const
24     {
25         return m_pos.get_row();
26     }
27
28     void move( char col, int row )
29     {
30         int curr_col = m_pos.get_col();
31         int curr_row = m_pos.get_row();
32         if ( (col != curr_col)
33             || (row != (curr_row + 1)) )
34             throw 42;
35         m_pos.move( col, row );
36     }
37
38 private:
39     Position m_pos;
40 };

```

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



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

```
1  class Integer : public IObject
2  {
3      public:
4          Integer()          { m_i = 0; }
5          Integer( int i ) { m_i = i; }
6
7          Integer* clone() const
8          {
9              return new Integer( *this );
10         }
11
12         bool eq( const IObject& obj ) const
13         {
14             const Integer* int_p = dynamic_cast<const Integer*>( &obj );
15             if ( int_p == nullptr )
16                 return false;
17             else
18                 return ( m_i == int_p->m_i );
19         }
20
21         bool lt( const IObject& obj ) const
22         {
23             const Integer* int_p = dynamic_cast<const Integer*>( &obj );
24             if ( int_p == nullptr )
25                 return false;
26             else
27                 return ( m_i < int_p->m_i );
28         }
29
30     private:
31         int m_i;
32     };
```

```
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 ints

```
1  class SListInt
2  {
3  public:
4      SListInt();
5      ~SListInt();
6      void push_front(
7          int v );
8      ...
9
10     class Itr
11     {
12     public:
13         Itr( Node* node_p );
14         void next();
15         int& get();
16         bool eq( Itr itr ) const;
17
18     private:
19         Node* m_node_p;
20     };
21
22     Itr begin();
23     Itr end();
24
25 private:
26     struct Node
27     {
28         int value;
29         Node* next_p;
30     };
31
32     Node* m_head_p;
33 };
```

- Object-oriented list which stores IObjects

```
1  class SListIObj
2  {
3  public:
4      SListIObj();
5      ~SListIObj();
6      void push_front(
7          const IObject& v );
8      ...
9
10     class Itr
11     {
12     public:
13         Itr( Node* node_p );
14         void next();
15         IObject&& get();
16         bool eq( Itr itr ) const;
17
18     private:
19         Node* m_node_p;
20     };
21
22     Itr begin();
23     Itr end();
24
25 private:
26     struct Node
27     {
28         IObject* obj_p;
29         Node* next_p;
30     };
31
32     Node* m_head_p;
33 };
```

4.2. Singly Linked List Implementation

- Implementation for singly linked list to store ints

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23

```

- Implementation for singly linked list to store IObjects

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23

```

```

1  SListInt::SListInt()
2  {
3      m_head_p = nullptr;
4  }
5
6  void SListInt::push_front( int v )
7
8  {
9      Node* new_node_p = new Node;
10     new_node_p->value = v;
11     new_node_p->next_p = m_head_p;
12     m_head_p = new_node_p;
13 }
14
15 SListInt::~SListInt()
16 {
17     while ( m_head_p != nullptr ) {
18         Node* temp_p = m_head_p->next_p;
19
20         delete m_head_p;
21         m_head_p = temp_p;
22     }
23 }

```

```

1  SListIObj::SListIObj()
2  {
3      m_head_p = nullptr;
4  }
5
6  void SListIObj::push_front(
7      const IObject& v )
8
9  {
10     Node* new_node_p = new Node;
11     new_node_p->obj_p = v.clone();
12     new_node_p->next_p = m_head_p;
13     m_head_p = new_node_p;
14 }
15
16 SListIObj::~SListIObj()
17 {
18     while ( m_head_p != nullptr ) {
19         Node* temp_p = m_head_p->next_p;
20         delete m_head_p->obj_p;
21         delete m_head_p;
22         m_head_p = temp_p;
23     }
24 }

```

```

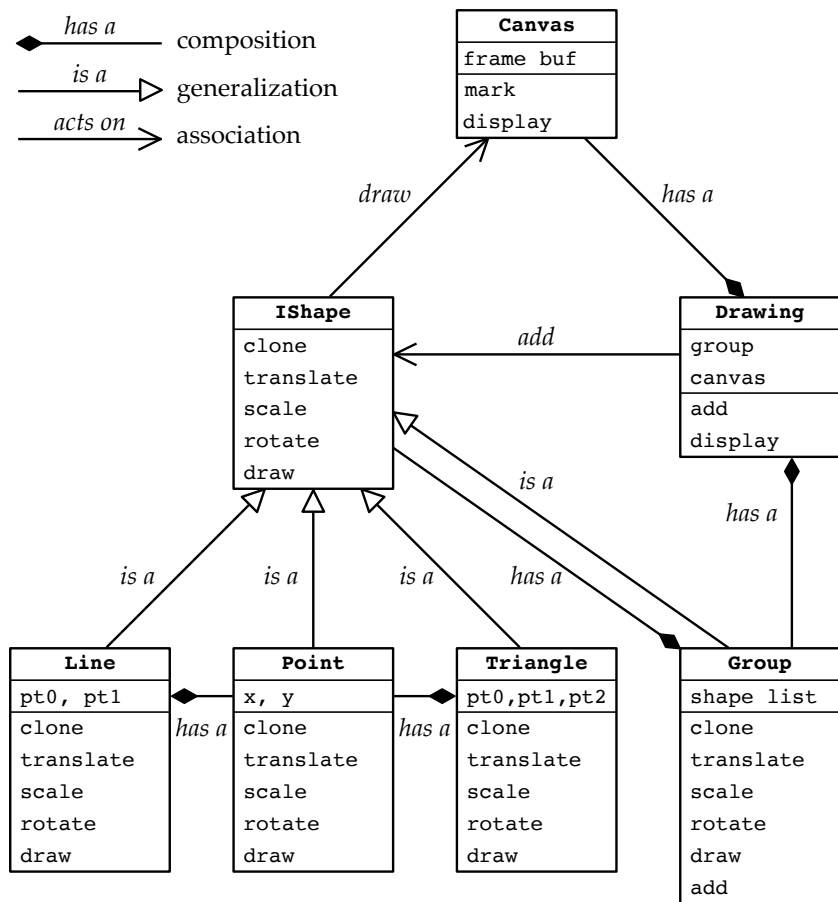
0000 01 class IObject
0000 02 {
0000 03     public:
0000 04         virtual IObject* clone( ) const = 0;
0000 05         ...
0000 06 };
0000 07
0000 08 class Integer : public IObject
0000 09 {
0000 10     public:
0000 11         Integer( int data ) { m_i = i; }
0000 12         Integer* clone() const {
0000 13             return new Integer( *this );
0000 14         }
0000 15         ...
0000 16     private:
0000 17         int m_i;
0000 18 };
0000 19
0000 20 SListIObj::SListIObj() {
0000 21     m_head_p = nullptr;
0000 22 }
0000 23
0000 24 void SListIObj::push_front(
0000 25     const IObject& v ) {
0000 26     Node* new_node_p = new Node;
0000 27     new_node_p->obj_p = v.clone();
0000 28     new_node_p->next_p = m_head_p;
0000 29     m_head_p = new_node_p;
0000 30 }
0000 31
0000 32 int main( void )
0000 33 {
0000 34     SListIObj list;
0000 35     Integer a(12);
0000 36     list.push_front(a);
0000 37     return 0;
0000 38 }

```

stack

heap

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

```
1  class Group : public IShape
2  {
3      public:
4          ...
5
6      ~Group() {
7          for ( int i = 0; i < m_shapes_size; i++ )
8              delete m_shapes[i];
9      }
10
11     void add( const IShape& shape ) {
12         assert( m_shapes_size < 16 );
13         m_shapes[m_shapes_size] = shape.clone();
14         m_shapes_size++;
15     }
16
17     void translate( double x_offset, double y_offset ) {
18         for ( int i = 0; i < m_shapes_size; i++ )
19             m_shapes[i]->translate( x_offset, y_offset );
20     }
21
22     void draw( Canvas* canvas ) const {
23         for ( int i = 0; i < m_shapes_size; i++ )
24             m_shapes[i]->draw( canvas );
25     }
26
27     private:
28         int      m_shapes_size;
29         IShape*  m_shapes[16];
30 };
```

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 };
```