

ECE 2400 Computer Systems Programming

Fall 2021

Topic 13: Object-Oriented Programming

School of Electrical and Computer Engineering
Cornell University

revision: 2021-08-28-22-26

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++ Resource Acquisition Is Initialization	24
1.6.	C++ Data Encapsulation	27
1.7.	C++ I/O Streams	28
2	Basic Object-Oriented Lists	31
2.1.	Singly Linked List Interface	31
2.2.	Singly Linked List Implementation	32
2.3.	Iterator-Based List Interface and Implementation	35
3	C++ Inheritance	39
3.1.	C++ Implementation Inheritance	41
3.2.	From Implementation to Interface Inheritance	45

3.3. C++ Interface Inheritance	52
3.4. Revisiting Composition vs. Generalization	55
4 Dynamic Polymorphic Object-Oriented Lists	57
4.1. Singly Linked List Interface	61
4.2. Singly Linked List Implementation	62
5 Drawing Framework Case Study	64

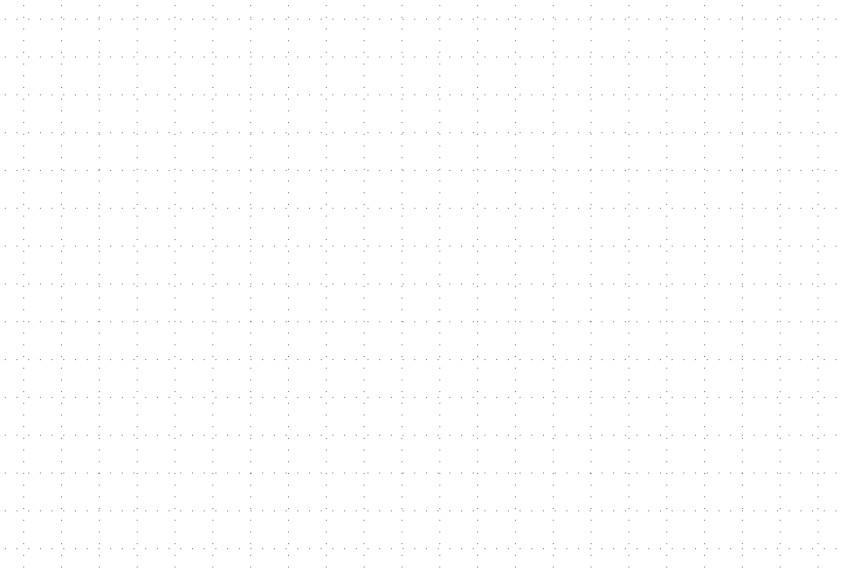
zyBooks The zyBooks logo is used to indicate additional material included in the course zyBook which will not be discussed in detail in lecture. Students are responsible for all material covered in lecture and in the course zyBook.

Copyright © 2021 Christopher Batten. All rights reserved. This handout was prepared by Prof. Christopher Batten at Cornell University for ECE 2400 / ENGRD 2140 Computer Systems Programming. 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.

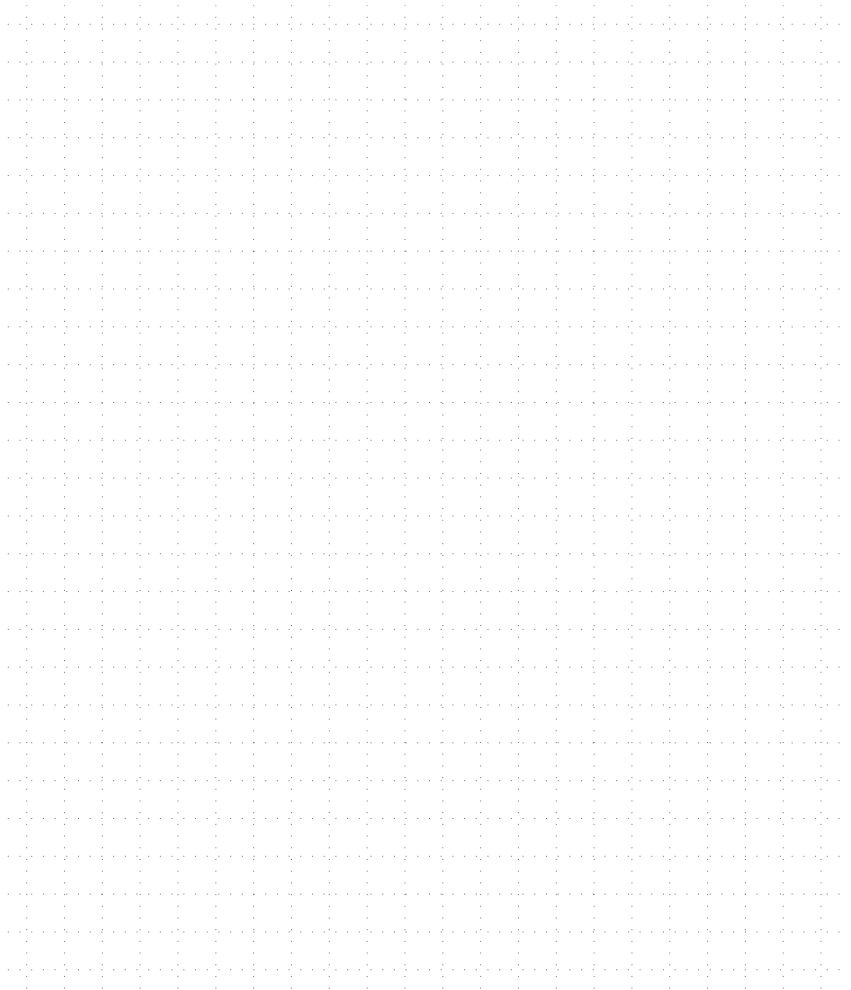
Object-oriented programming

- 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

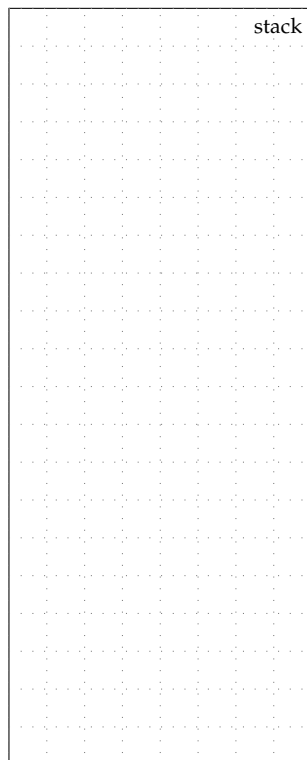


1. C++ Classes

- 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 }
30
31 void point_print( point_t* this )
32 {
33     std::printf("%.2f,%.2f", this->x, this->y );
34 }
```

```
□□□ 01 int main( void )
□□□ 02 {
□□□ 03     point_t pt;
□□□ 04     pt.x = 0;
□□□ 05     pt.y = 0;
□□□ 06
□□□ 07     point_translate( &pt, 1, 0 );
□□□ 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
26     void point_print() const
27     {
28         std::printf("%.2f,%.2f", this->x, this->y );
29     }
30 };
```

- 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, 0 );
□□□ 08     pt.point_scale( 2 );
□□□ 09     return 0;
□□□ 10 }

```

- Recall object-oriented prog in C

```

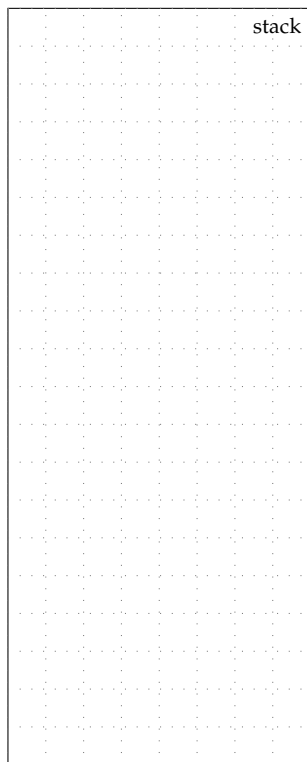
1  int main( void )
2  {
3      point_t pt;
4      pt.x = 0;
5      pt.y = 0;
6
7      point_translate( &pt, 1, 2 );
8      point_scale    ( &pt, 2 );
9      return 0;
10 }

```

```

point_translate( &pt, 1, 0 ) ↔ pt.point_translate( 1, 0 )
foo( &bar, ... ) ↔ bar.foo( ... )

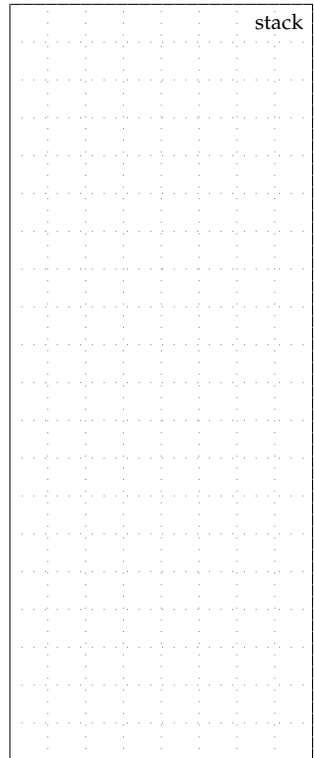
```



- 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
26     void print() const
27     {
28         std::printf("%.2f,%.2f", x, y );
29     }
30 };
```

```
□□□ 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

```
0001 struct Point
0002 {
0003     double x;
0004     double y;
0005
0006     // default constructor
0007     Point()
0008     {
0009         x = 0.0;
0010         y = 0.0;
0011     }
0012
0013     // non-default constructor
0014     Point( double x_, double y_ )
0015     {
0016         x = x_;
0017         y = y_;
0018     }
0019
0020     ...
0021 };
0022
0023 int main( void )
0024 {
0025     Point pt0;
0026     Point pt1( 1, 2 );
0027     pt0 = pt1;
0028     pt0.translate( 1, 0 );
0029     return 0;
0030 }
```

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
```

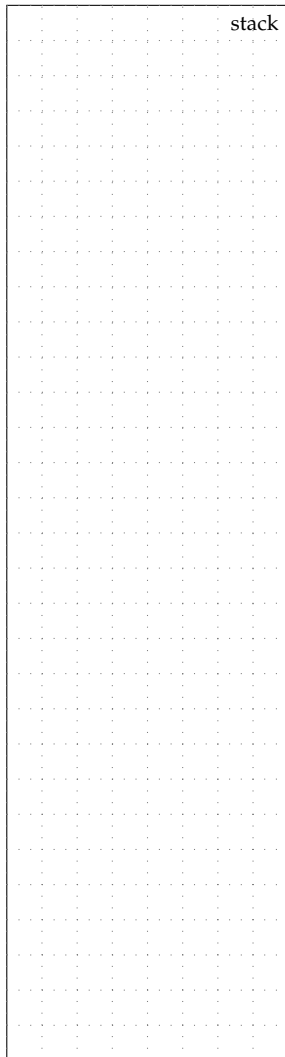
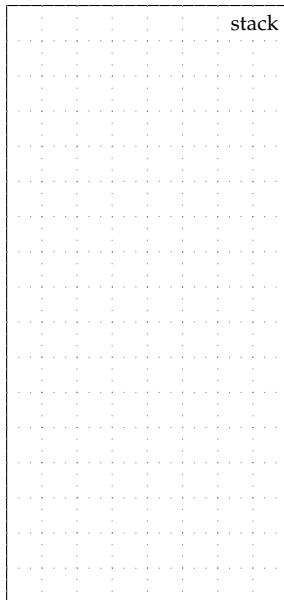
- Initialization lists initialize members before body of constructor
 - Avoids creating a temporary default object
 - Required for initializing reference members
 - Prefer initialization lists when possible

```
1 struct Point                1 struct Point
2 {                            2 {
3     double x;                3     double x;
4     double y;                4     double y;
5
6     // default constructor    6     // default constructor
7     Point()                  7     Point()
8     { x = 0.0; y = 0.0; }    8     : x(0.0), y(0.0) { }
9
10    // non-default constructor 10    // non-default constructor
11    Point( double x_, double y_ ) 11    Point( double x_, double y_ )
12    { x = x_; y = y_; }      12    : x(x_), y(y_) { }
13
14    ...                       14    ...
15 };                          15 };
```

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)

```
0001 Point operator+( const Point& pt0,  
0002                   const Point& pt1 )  
0003 {  
0004     Point tmp = pt0;  
0005     tmp.translate( pt1.x, pt1.y );  
0006     return tmp;  
0007 }  
0008  
0009 int main( void )  
0010 {  
0011     Point ptA(1,2);  
0012     Point ptB(3,4);  
0013     Point ptC = ptA + ptB;  
0014     return 0;  
0015 }
```



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

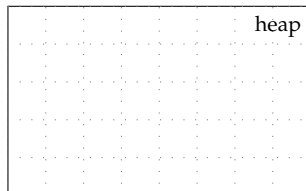
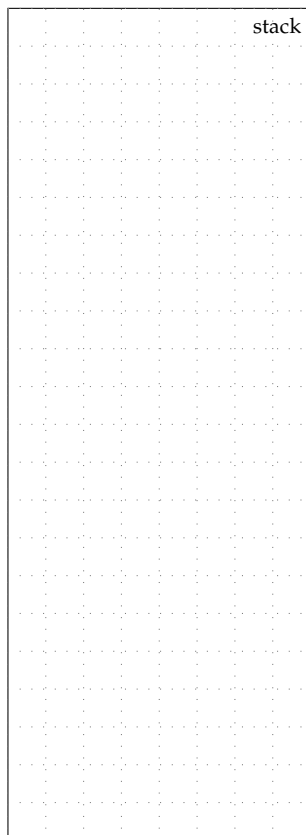
1.4. C++ Rule of Three

- 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 = 0.0;
10         *y_p = 0.0;
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, 0 );
26     return 0;
27 }

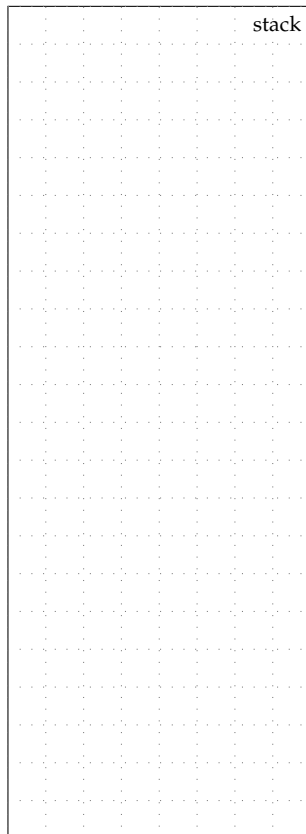
```



C++ Destructors

- Destructors are special member functions for destroying an object

```
□□□ 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 = 0.0;
□□□ 10         *y_p = 0.0;
□□□ 11     }
□□□ 12
□□□ 13     ~DPoint() {
□□□ 14         delete x_p;
□□□ 15         delete y_p;
□□□ 16     }
□□□ 17     ...
□□□ 18 };
□□□ 19
□□□ 20 int main( void )
□□□ 21 {
□□□ 22     DPoint pt0;
□□□ 23     pt0.translate( 1, 0 );
□□□ 24     return 0;
□□□ 25 }
```

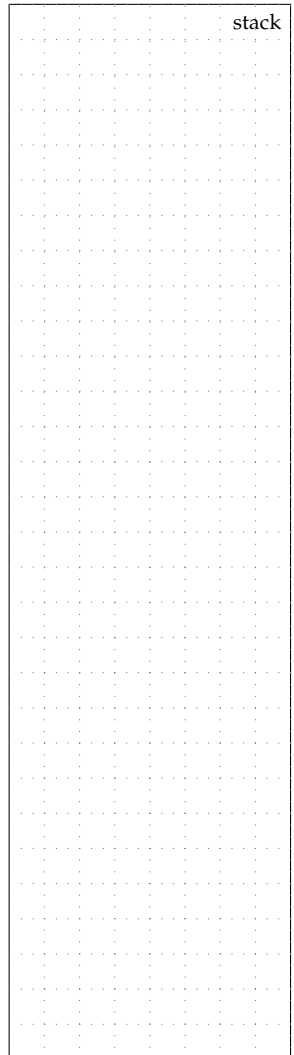


- What if we copy an object with dynamically allocated memory?

```

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 = 0.0;
0000 10         *y_p = 0.0;
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, 0 );
0000 24     return 0;
0000 25 }

```



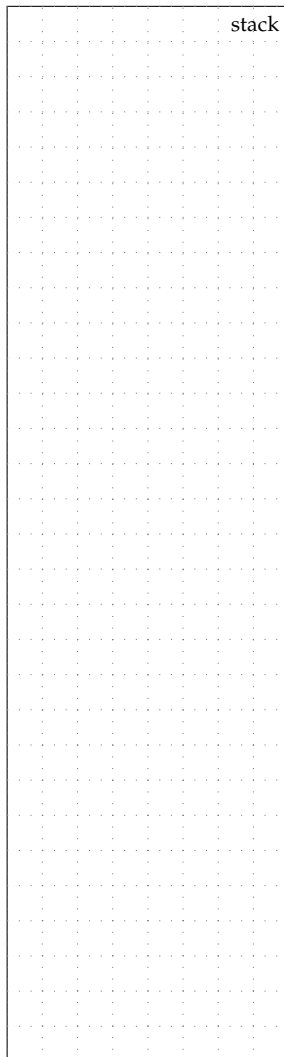
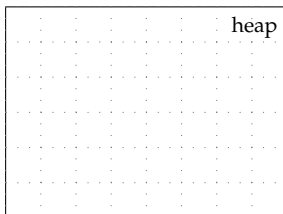
C++ Copy Constructors

- Copy constructors are special member functions for constructing a new object from an old object

```

0000 01 struct DPoint
0000 02 {
0000 03     double* x_p;
0000 04     double* y_p;
0000 05
0000 06     DPoint( const DPoint& pt ) {
0000 07         x_p = new double;
0000 08         y_p = new double;
0000 09         *x_p = *pt.x_p;
0000 10         *y_p = *pt.y_p;
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, 0 );
0000 24     return 0;
0000 25 }

```

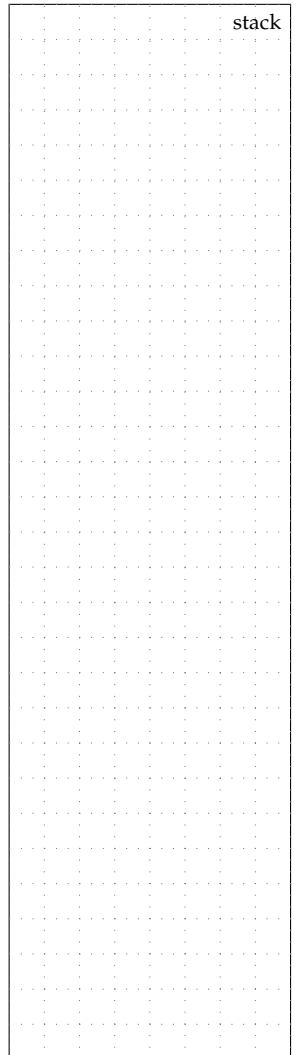
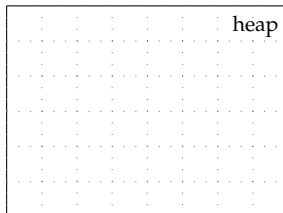


- What if we assign to an object with dynamically allocated memory?

```

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 = 0.0;
10         *y_p = 0.0;
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, 0 );
25     return 0;
26 }

```



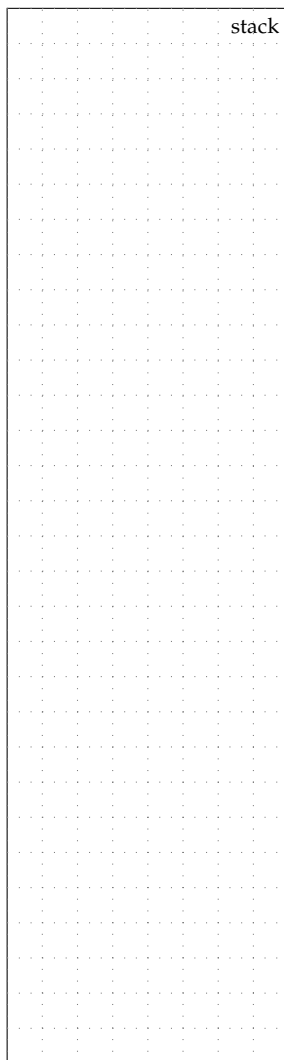
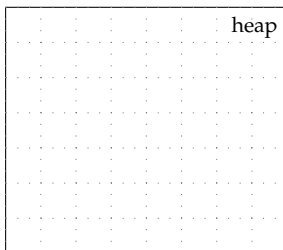
C++ Assignment Operators

- An overloaded assignment operator will be called for assignment

```

0001 struct DPoint
0002 {
0003     double* x_p;
0004     double* y_p;
0005
0006     DPoint&
0007     operator=( const DPoint& pt )
0008     {
0009         if ( this != &pt ) {
0010             *x_p = *pt.x_p;
0011             *y_p = *pt.y_p;
0012         }
0013         return *this;
0014     }
0015     ...
0016 };
0017
0018 int main( void )
0019 {
0020     DPoint pt0;
0021     DPoint pt1;
0022     pt1 = pt0;
0023     pt0.translate( 1, 0 );
0024     return 0;
0025 }

```



C++ Rule of Three

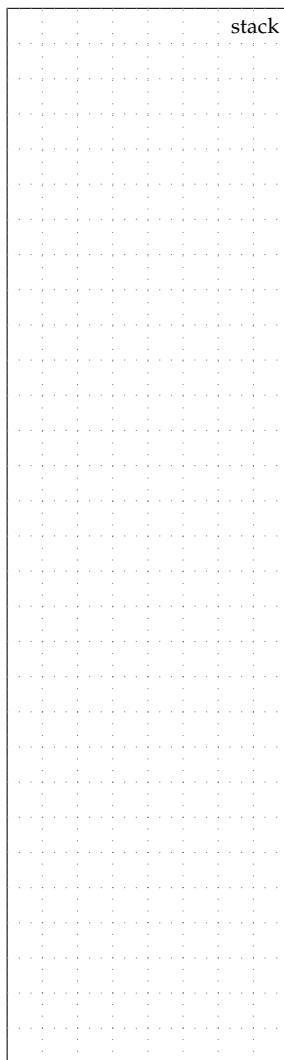
- Default **destructor**, **copy constructor**, and **assignment operator** will work fine for simple classes
- For a more complex class may need to define one of these ...
- ... and if you define one, then you probably need to define all three!
- Be very careful about self assignment

```
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 = 0.0;
11         *y_p = 0.0;
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     }
21
22     DPoint( const DPoint& pt )
23     {
24         x_p = new double;
25         y_p = new double;
26         *x_p = *pt.x_p;
27         *y_p = *pt.y_p;
28     }
29
30     ~DPoint()
31     {
32         delete x_p;
33         delete y_p;
34     }
35
36     DPoint&
37     operator=( const DPoint& pt )
38     {
39         if ( this != &pt ) {
40             *x_p = *pt.x_p;
41             *y_p = *pt.y_p;
42         }
43         return *this;
44     }
45
46     ...
47 };
```

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++ Resource Acquisition Is Initialization

- What if we forget to call `delete`? What if there is an exception?
- RAII is a design pattern that ties a resource to object lifetime (also known as **scope-bound resource management**)
- 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 char* get_first_word( const char* str )
2 {
3     // Determine how many characters are in first word
4     int len = 0;
5     while ( (str[len] != ' ') && (str[len] != '\0') )
6         len++;
7
8     // Dynamically allocate space for first word
9     char* first_word = new char[len+1];
10
11    // Copy the first word to the new allocated string
12    for ( int i = 0; i < len; i++ )
13        first_word[i] = str[i];
14    first_word[len] = '\0';
15
16    return first_word;
17 }
18
19 int main( void )
20 {
21     char* x = get_first_word( "foo bar" );
22     char* y = get_first_word( "foo" );
23     char* z = get_first_word( "foo bar baz" );
24     return 0;
25 };
```



```
1  struct String
2  {
3      String()                { m_str = nullptr;          }
4      String( int size )     { m_str = new char[size];    }
5      char& operator[] ( int idx ) { return m_str[idx];    }
6      ~String()              { delete[] m_str;           }
7      String( const String& str ) { copy( str.m_str );    }
8      const char* c_str() const { return m_str;         }
9
10     String& operator=( const String& str )
11     {
12         if ( this != &str ) {
13             delete[] m_str;
14             copy( str.m_str );
15         }
16         return *this;
17     }
18
19     void copy( const char* s )
20     {
21         // Determine how many characters are in given string
22         int len = 0;
23         while ( s[len] != '\0' )
24             len++;
25
26         // Dynamically allocate space for copy
27         m_str = new char[len+1];
28
29         // Copy the given string to the newly allocated string
30         for ( int i = 0; i < len+1; i++ )
31             m_str[i] = s[i];
32     }
33
34     char* m_str;
35 };
```

```
1 String get_first_word( const char* str )
2 {
3     int len = 0;
4     while ( (str[len] != ' ') && (str[len] != '\0') )
5         len++;
6
7     String first_word(len+1);
8
9     for ( int i = 0; i < len; i++ )
10        first_word[i] = str[i];
11
12    first_word[len] = '\0';
13    return first_word;
14 }
15
16 int main( void )
17 {
18     String x = get_first_word( "foo bar" );
19     String y = get_first_word( "foo" );
20     String z = get_first_word( "foo bar baz" );
21     return 0;
22 };
```

- Every time you use a new think critically ...
- ... who is in charge of calling delete?
- ... what happens if there is an exception through?
- Every use of new/delete needs to be purposeful and only if absolutely necessary!

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 x; double y;
5
6     public:
7         // default constructor
8         Point() { x = 0.0; y = 0.0; }
9
10        // non-default constructor
11        Point( double x_, double y_ ) { x = x_; 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

1.7. C++ I/O Streams

- `printf` does not support user-defined types
- Need to encapsulate printing in a member function ...
- ... but this leads to very awkward syntax

```
1 int main( void )
2 {
3     Point pt0(1,2);
4     Point pt1 = pt0;
5     pt1.translate(2.0,2.0);
6
7     std::printf("initial point = ");
8     pt0.print();
9     std::printf("\n");
10    std::printf("translate by %.2f,%.2f\n", 2.0, 2.0 );
11    std::printf("new point = ");
12    pt1.print();
13    std::printf("\n");
14 }
```

- Clever use of operator overloading can provide cleaner syntax

```
1  struct ostream
2  {
3      // internal stream state
4  };
5
6  ostream cout;
7
8  ostream& operator<<( ostream& os, int i )
9  {
10     std::printf("%d",i); return os;
11 }
12
13 ostream& operator<<( ostream& os, double d )
14 {
15     std::printf("%.2f",d); return os;
16 }
17
18 ostream& operator<<( ostream& os, const char* str )
19 {
20     std::printf("%s",str); return os;
21 }
22
23 ostream& operator<<( ostream& os, const Point& pt )
24 {
25     pt.print(); return os;
26 }
27
28 struct EndOfLine
29 { };
30
31 EndOfLine endl;
32
33 ostream& operator<<( ostream& os, const EndOfLine& endl )
34 {
35     std::printf("\n"); return os;
36 }
```

```
1  int main( void )
2  {
3      Point pt0(1,2);
4      Point pt1 = pt0;
5      pt1.translate(2.0,2.0);
6
7      cout << "initial point = " << pt0 << endl;
8      cout << "translate by " << 2.0 << ", " << 2.0 << endl;
9      cout << "new point = " << pt1 << endl;
10 }
```

- The standard C++ library provides powerful streams
 - `iostream` : write/read standard I/O as streams
 - `fstream` : write/read files as streams
 - `sstream` : write/read strings as streams

2. Basic Object-Oriented Lists

- 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

```
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 void slist_int_reverse ( slist_int_t* this );
```

- 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         void reverse();       // member function
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         void reverse(); // member function
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

```

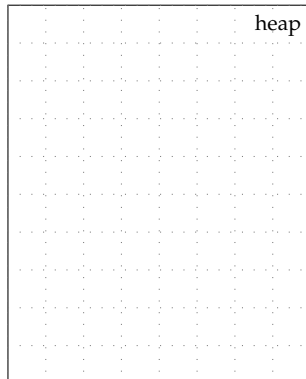
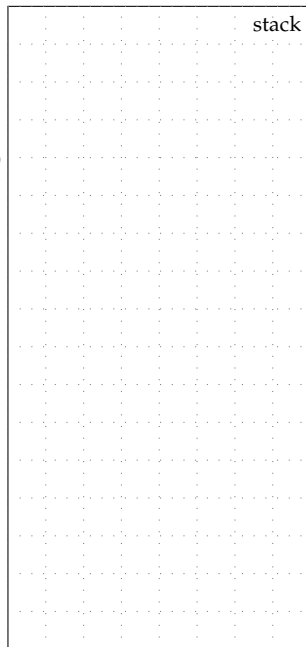
1 SListInt::SListInt()
2
3 {
4     m_head_p = nullptr;
5 }
6
7 void SListInt::push_front( int v )
8
9 {
10    Node* new_node_p
11    = new Node;
12    new_node_p->value = v;
13    new_node_p->next_p = m_head_p;
14    m_head_p = new_node_p;
15 }
16
17 SListInt::~SListInt()
18
19 {
20     while ( m_head_p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

```

00000001 SListInt::SListInt()
00000002 {
00000003     m_head_p = nullptr;
00000004 }
00000005
00000006 void SListInt::push_front( int v )
00000007 {
00000008     Node* new_node_p
00000009         = new Node;
00000010     new_node_p->value = v;
00000011     new_node_p->next_p = m_head_p;
00000012     m_head_p = new_node_p;
00000013 }
00000014
00000015 int main( void )
00000016 {
00000017     SListInt lst;
00000018     lst.push_front(12);
00000019     lst.push_front(11);
00000020     lst.push_front(10);
00000021
00000022     SListInt::Node* node_p
00000023         = lst.m_head_p;
00000024     while ( node_p != nullptr ) {
00000025         int value = node_p->value;
00000026         node_p = node_p->next_p;
00000027     }
00000028
00000029     return 0;
00000030 }

```



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
5 void SListInt::Itr::next()
6 {
7     assert( m_node_p != nullptr );
8     m_node_p = m_node_p->next_p;
9 }
10
11 int& SListInt::Itr::get()
12 {
13     assert( m_node_p != nullptr );
14     return m_node_p->value;
15 }
16
17 bool SListInt::Itr::eq( Itr itr ) const
18 {
19     return ( m_node_p == itr.m_node_p );
20 }
21
22 SListInt::Itr SListInt::begin()
23 {
24     return Itr(m_head_p);
25 }
26
27 SListInt::Itr SListInt::end()
28 {
29     return Itr(nullptr);
30 }
31
32 SListInt::Node* node_p          1 SListInt::Itr itr
33   = list.m_head_p;             2   = list.begin();
34 while ( node_p != nullptr ) {  3 while ( !itr.eq(list.end()) ) {
35     int value = node_p->value   4     int value = itr.get();
36     printf( "%d\n", value );   5     printf( "%d\n", value );
37     node_p = node_p->next_p;   6     itr.next();
38 }                               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     Pawn( char col, int row )
5         : m_col(col), m_row(row)
6     { }
7
8     int get_pts() const
9     {
10         return 1;
11     }
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         if ( (col != m_col)
26             || (row != (m_row + 1)) )
27             throw 42;
28
29         m_col = col;
30         m_row = row;
31     }
32
33     void print() const
34     {
35         std::printf( "pawn@%c%d",
36                     m_col, m_row );
37     }
38
39 private:
40     char m_col;
41     int m_row;
42 };
```

```
1 class Rook
2 {
3 public:
4     Rook( char col, int row )
5         : m_col(col), m_row(row)
6     { }
7
8     int get_pts() const
9     {
10         return 5;
11     }
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         if ( (col != m_col)
26             && (row != m_row) )
27             throw 42;
28
29         m_col = col;
30         m_row = row;
31     }
32
33     void print() const
34     {
35         std::printf( "rook@%c%d",
36                     m_col, m_row );
37     }
38
39 private:
40     char m_col;
41     int m_row;
42 };
```

- 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



```

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
20
21
22
23
24    void move( char col, int row )
25    {
26        m_col = col;
27        m_row = row;
28    }
29
30
31
32
33
34
35    // derived classes cannot
36    // access private data in
37    // base class
38    protected:
39
40
41    char m_col;
42    int m_row;
43
44 };

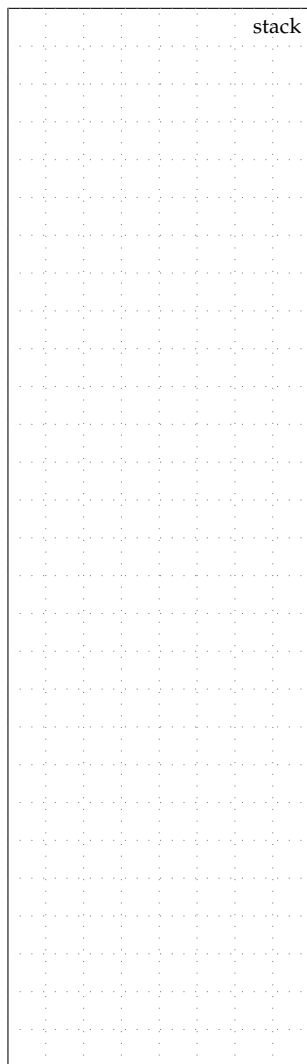
```

```

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    int get_pts() const
20    {
21        return 5;
22    }
23
24    void move( char col, int row )
25    {
26        if ( (col != m_col)
27            && (row != m_row) )
28            throw 42;
29
30        Piece::move( col, row );
31    }
32
33    void print() const
34    {
35        std::printf( "rook%c%d",
36                    m_col, m_row );
37    }
38
39
40
41
42
43
44 };

```

```
0000 01 class Piece
0000 02 {
0000 03     Piece( char col, int row )
0000 04         : m_col(col), m_row(row)
0000 05     { }
0000 06
0000 07     char get_col() const
0000 08     {
0000 09         return m_col;
0000 10     }
0000 11
0000 12     int get_row() const
0000 13     {
0000 14         return m_row;
0000 15     }
0000 16     ...
0000 17 };
0000 18
0000 19 class Pawn : public Piece
0000 20 {
0000 21     Pawn( char col, int row )
0000 22         : Piece( col, row )
0000 23     { }
0000 24
0000 25     int get_pts() const
0000 26     {
0000 27         return 1;
0000 28     }
0000 29     ...
0000 30 };
0000 31
0000 32 int main( void )
0000 33 {
0000 34     Pawn p('a',2);
0000 35     char col = p.get_col();
0000 36     return 0;
0000 37 }
```



```
01 class Piece
02 {
03     Piece( char col, int row )
04         : m_col(col), m_row(row)
05     { }
06
07     char get_col() const
08     {
09         return m_col;
10     }
11
12     int get_row() const
13     {
14         return m_row;
15     }
16     ...
17 };
18
19 class Pawn : public Piece
20 {
21     int get_pts() const { return 1; }
22     ...
23 };
24
25 class Rook : public Piece
26 {
27     int get_pts() const { return 5; }
28     ...
29 };
30
31 int main( void )
32 {
33     Pawn p('a',2);
34     Rook r('h',3);
35     int pts0 = p.get_pts();
36     int pts1 = r.get_pts();
37     int sum  = pts0 + pts1;
38     return 0;
39 }
```



stack

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 }
```

```
0001 class Piece
0002 {
0003     Piece( char col, int row )    { ...
0004     char get_col() const        { ...
0005     int  get_row() const        { ...
0006     void move( char col, int row ) { ...
0007     ...
0008 };
0009
0010 class Pawn : public Piece
0011 {
0012     int get_pts() const { return 1; }
0013     ...
0014 };
0015
0016 class Rook : public Piece
0017 {
0018     int get_pts() const { return 5; }
0019     ...
0020 };
0021
0022 int calc_pts( Piece* p0, Piece* p1 )
0023 {
0024     int pts0 = p0->get_pts();
0025     int pts1 = p1->get_pts();
0026     return pts0 + pts1;
0027 }
0028
0029 int main( void )
0030 {
0031     Pawn p('a',2);
0032     Rook r('h',3);
0033     int sum = calc_pts( &p, &r );
0034     return 0;
0035 }
```

stack

```

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         : m_type(type),
12           m_col(col), m_row(row)
13     { }
14
15     PieceType get_type() const
16     {
17         return m_type;
18     }
19
20     char get_col() const
21     {
22         return m_col;
23     }
24
25     int get_row() const
26     {
27         return m_row;
28     }
29
30     void move( char col, int row )
31     {
32         m_col = col;
33         m_row = row;
34     }
35
36     protected:
37
38     // explicit type field
39     PieceType m_type;
40
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    void print() const
24    {
25        std::printf( "rook%c%d",
26                    m_col, m_row );
27    }
28 };

```



```
0001 class Pawn : public Piece
0002 {
0003     int get_pts() const { return 1; }
0004     ...
0005 };
0006
0007 int calc_pts( Piece* p0, Piece* p1 )
0008 {
0009     int pts0;
0010     if ( p0->get_type() == PAWN ) {
0011         Pawn* pawn_p = (Pawn*) p0;
0012         pts0 = pawn_p->get_pts();
0013     }
0014     else if ( p0->get_type() == ROOK ) {
0015         Rook* rook_p = (Rook*) p0;
0016         pts0 = rook_p->get_pts();
0017     }
0018
0019     int pts1;
0020     if ( p1->get_type() == PAWN ) {
0021         Pawn* pawn_p = (Pawn*) p1;
0022         pts1 = pawn_p->get_pts();
0023     }
0024     else if ( p1->get_type() == ROOK ) {
0025         Rook* rook_p = (Rook*) p1;
0026         pts1 = rook_p->get_pts();
0027     }
0028
0029     return pts0 + pts1;
0030 }
0031
0032 int main( void )
0033 {
0034     Pawn p('a',2);
0035     Rook r('h',3);
0036     int sum = calc_pts( &p, &r );
0037     return 0;
0038 }
```

stack

```

1  enum PieceType { PAWN, ROOK };
2
3  class Piece
4  {
5  public:
6
7      Piece( PieceType type,
8             char col, int row )
9          : m_type(type),
10            m_col(col), 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     void print() const
24     {
25         std::printf( "rook%c%d",
26                     m_col, m_row );
27     }
28 };

```

```
01 class Piece
02 {
03     ...
04     int get_pts() const
05     {
06         if ( m_type == PAWN ) {
07             Pawn* pawn_p = (Pawn*) this;
08             return pawn_p->get_pts();
09         }
10         else if ( m_type == ROOK ) {
11             Rook* rook_p = (Rook*) this;
12             return rook_p->get_pts();
13         }
14     }
15 };
16
17 class Pawn : public Piece
18 {
19     int get_pts() const { return 1; }
20     ...
21 };
22
23 int calc_pts( Piece* p0, Piece* p1 )
24 {
25     int pts0 = p0->get_pts();
26     int pts1 = p1->get_pts();
27     return pts0 + pts1;
28 }
29
30 int main( void )
31 {
32     Pawn p('a',2);
33     Rook r('h',3);
34     int sum = calc_pts( &p, &r );
35     return 0;
36 }
```

stack

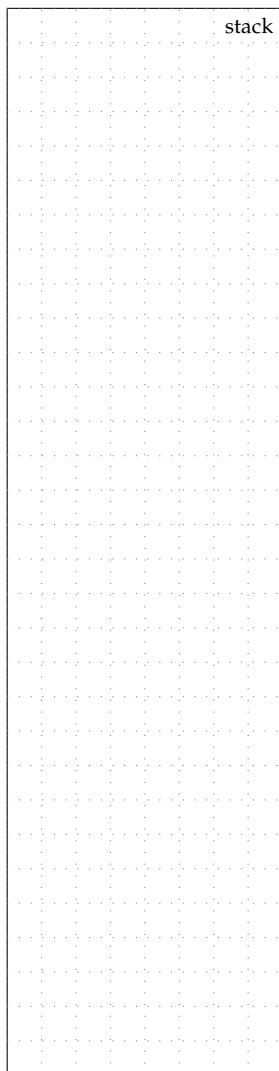
3.3. C++ Interface Inheritance

- 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         : m_col(col), m_row(row)
7     { }
8
9     // pure virtual function
10    virtual int get_pts() const = 0;
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        m_col = col;
25        m_row = row;
26    }
27
28    protected:
29    char    m_col;
30    int    m_row;
31 };

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    void print() const
24    {
25        std::printf( "rook%c%d",
26                    m_col, m_row );
27    }
28 };
```

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

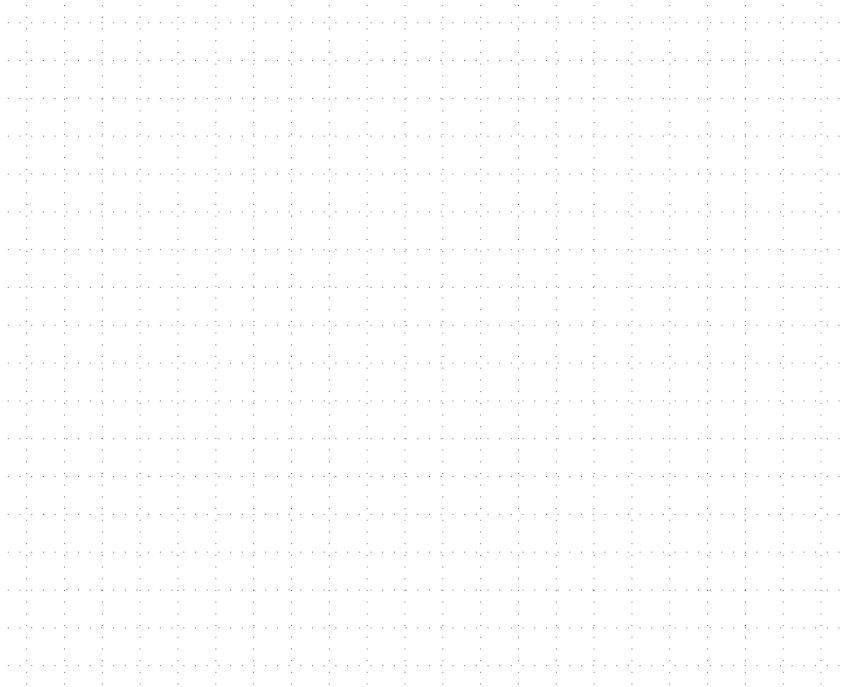


Abstract base classes

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

```
1  class IPiece
2  {
3  public:
4      virtual ~IPiece() { }
5      virtual int  get_pts() const = 0;
6      virtual char get_col() const = 0;
7      virtual int  get_row() const = 0;
8      virtual void move( char col, int row ) = 0;
9      virtual void print() const = 0;
10 };
11
12 class Rook : public IPiece
13 {
14 public:
15     Rook( char col, int row ) : m_col(col), m_row(row) { }
16     ~Rook() { }
17
18     int  get_pts() const { return 5;    }
19     char get_col() const { return m_col; }
20     int  get_row() const { return m_row; }
21
22     void move( char col, int row )
23     {
24         if ( col != m_col) && (row != m_row) )
25             throw 42;
26         m_col = col;
27         m_row = row;
28     }
29
30     void print() const
31     {
32         std::printf( "rook%c%d", m_col, m_row );
33     }
34
35 private:
36     char m_col;
37     int  m_row;
38 };
```

3.4. Revisiting Composition vs. Generalization



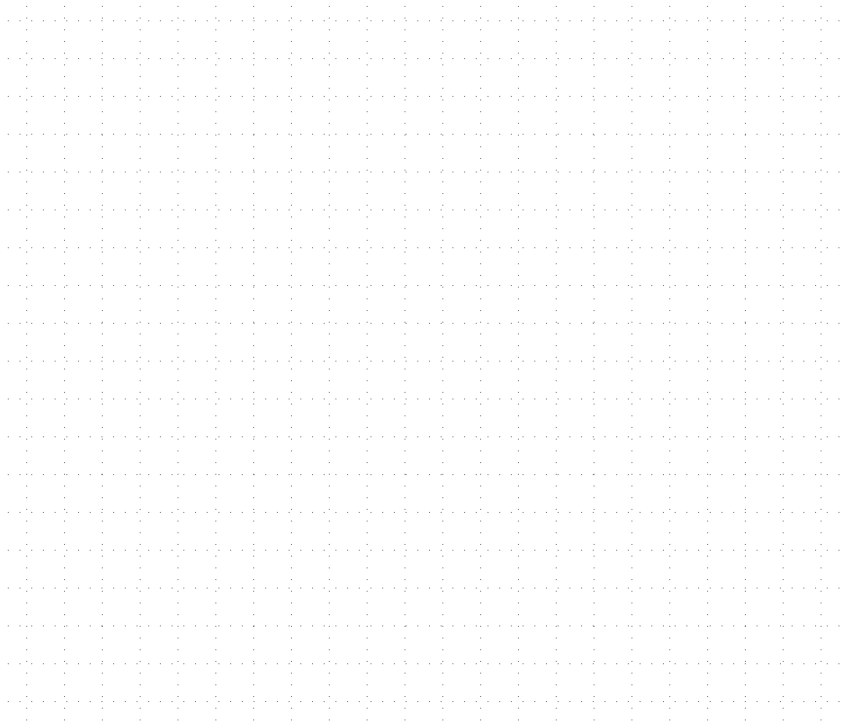
```
1 class Position
2 {
3     public:
4         Position( char col, int row )
5             : m_col(col), m_row(row)
6         { }
7
8         char get_col() const
9         {
10             return m_col;
11         }
12
13         int get_row() const
14         {
15             return m_row;
16   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

```
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         int get_pts() const
12         {
13             return 1;
14         }
15
16         char get_col() const
17         {
18             return m_pos.get_col();
19         }
20
21         int get_row() const
22         {
23             return m_pos.get_row();
24         }
25
26         void move( char col, int row )
27         {
28             int curr_col = m_pos.get_col();
29             int curr_row = m_pos.get_row();
30             if ( (col != curr_col)
31                 || (row != (curr_row + 1)) )
32                 throw 42;
33             m_pos.move( col, row );
34         }
35
36         void print() const
37         {
38             std::printf("pawn@");
39             m_pos.print();
40         }
41
42     private:
43         Position m_pos;
44 };
```


4. Dynamic Polymorphic Object-Oriented Lists

- 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



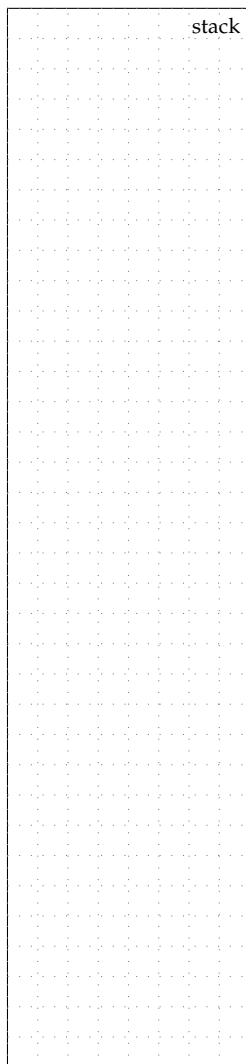
```
1  class IObject
2  {
3  public:
4
5      virtual ~IObject() { };
6      virtual IObject*   clone( )           const = 0;
7      virtual bool      eq( const IObject& rhs ) const = 0;
8      virtual bool      lt( const IObject& rhs ) const = 0;
9      virtual void      print()           const = 0;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

```
1  class Integer : public IObject
2  {
3  public:
4      Integer()          : m_data(0)          { }
5      Integer( int data ) : m_data(data)      { }
6
7      Integer* clone() const
8      {
9          return new Integer( *this );
10     }
11
12     bool eq( const IObject& rhs ) const
13     {
14         const Integer* rhs_p = dynamic_cast<const Integer*>( &rhs );
15         if ( rhs_p == nullptr )
16             return false;
17         else
18             return ( m_data == rhs_p->m_data );
19     }
20
21     bool lt( 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     void print() const
31     {
32         std::printf( "%d", m_data );
33     }
34
35     private:
36         int m_data;
37 };
```

```

0000 01 class IObject
0000 02 {
0000 03     public:
0000 04         virtual bool
0000 05             eq( const IObject& rhs ) const = 0;
0000 06         ...
0000 07 };
0000 08
0000 09 bool operator==( const IObject& lhs,
0000 10                  const IObject& rhs ) {
0000 11     return lhs.eq(rhs);
0000 12 }
0000 13
0000 14 class Integer : public IObject
0000 15 {
0000 16     public:
0000 17         Integer( int data ) : m_data(data) { }
0000 18
0000 19         bool eq( const IObject& rhs ) const {
0000 20             const Integer* rhs_p
0000 21                 = dynamic_cast<const Integer*>(&rhs);
0000 22             if ( rhs_p == nullptr )
0000 23                 return false;
0000 24             else
0000 25                 return (m_data == rhs_p->m_data);
0000 26         }
0000 27         ...
0000 28     private:
0000 29         int m_data;
0000 30 };
0000 31
0000 32 int main( void )
0000 33 {
0000 34     Integer a(2);
0000 35     Integer b(3);
0000 36     bool c = ( a == b );
0000 37     return 0;
0000 38 }

```



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      void reverse();
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      void reverse();
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

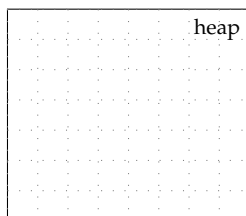
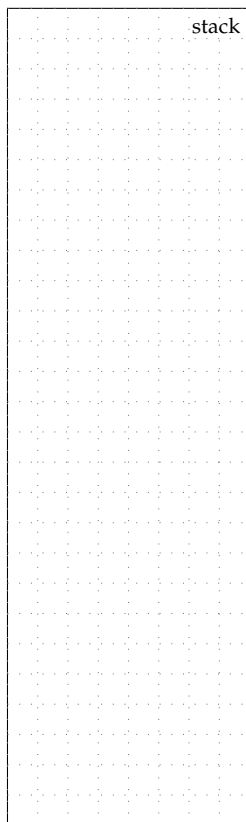
- Implementation for singly linked list to store ints

- Implementation for singly linked list to store IObjects

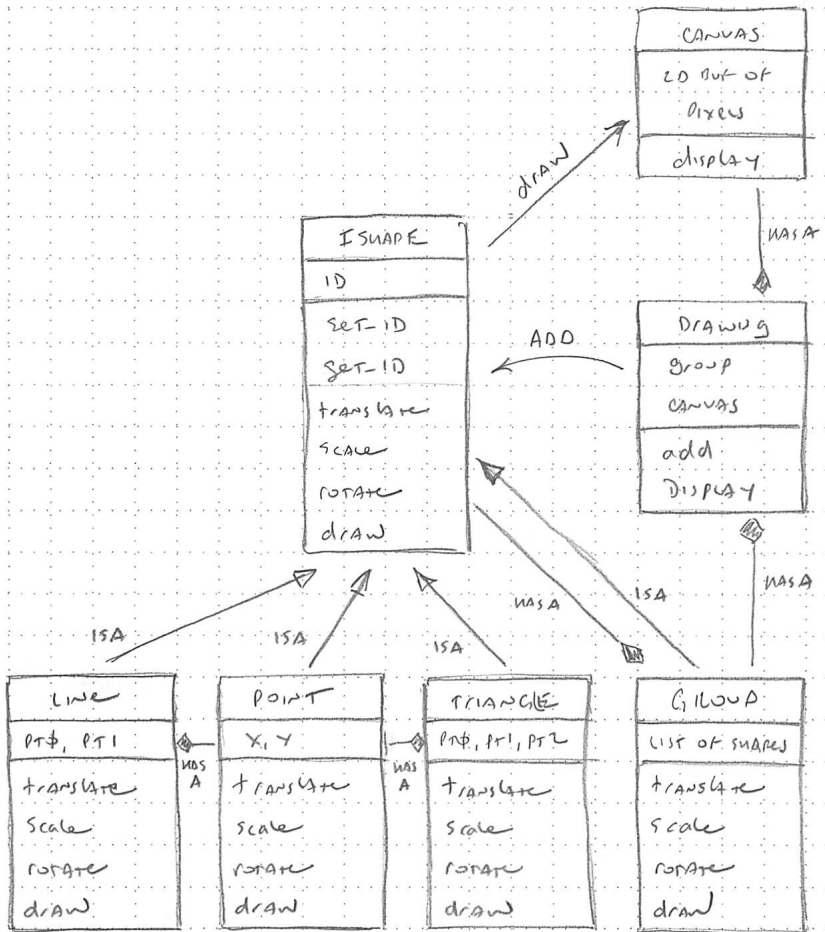
```
1 SListInt::SListInt()
2 {
3     m_head_p = nullptr;
4 }
5
6 void SListInt::push_front( int v )
7 {
8     Node* new_node_p = new Node;
9     new_node_p->value = v;
10    new_node_p->next_p = m_head_p;
11    m_head_p = new_node_p;
12 }
13
14 SListInt::~SListInt()
15 {
16     while ( m_head_p != nullptr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

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

```
000001 class IObject
000002 {
000003     public:
000004         virtual IObject* clone( ) const = 0;
000005         ...
000006 };
000007
000008 class Integer : public IObject
000009 {
000010     public:
000011     Integer( int data ) : m_data(data) { }
000012     Integer* clone() const {
000013         return new Integer( *this );
000014     }
000015     ...
000016     private:
000017     int m_data;
000018 };
000019
000020 SListIObj::SListIObj() {
000021     m_head_p = nullptr;
000022 }
000023
000024 void SListIObj::push_front(
000025     const IObject& v ) {
000026     Node* new_node_p = new Node;
000027     new_node_p->obj_p = v.clone();
000028     new_node_p->next_p = m_head_p;
000029     m_head_p = new_node_p;
000030 }
000031
000032 int main( void )
000033 {
000034     SListIObj list;
000035     Integer a(12);
000036     list.push_front(a);
000037     return 0;
000038 }
```



5. Drawing Framework Case Study



<https://repl.it/@cbatten/ece2400-T13-ex3>