# 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.

## 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

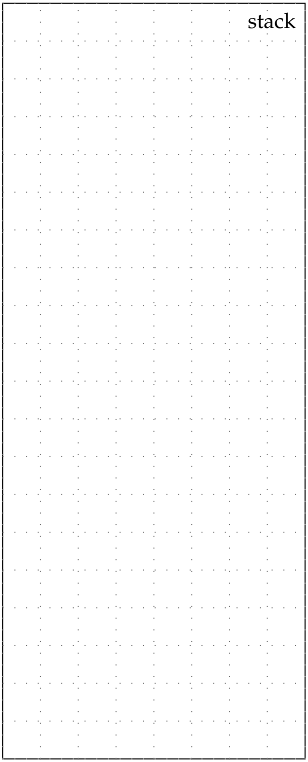
## Example class diagram for shapes and drawings

# 1. C++ Classes

- Perfectly possible to use object-oriented programming in C

```
typedef struct
{
  double x;
  double y;
}
point_t;

void point_translate( point_t* this,
                      double x_offset, double y_offset )
{
  this->x += x_offset; this->y += y_offset;
}

void point_scale( point_t* this, double factor )
{
  this->x *= factor; this->y *= factor;
}

void point_rotate( point_t* this, double angle )
{
  const double pi = 3.14159265358979323846;
  double s = std::sin((angle*pi)/180);
  double c = std::cos((angle*pi)/180);

  double x_new = (c * this->x) - (s * this->y);
  double y_new = (s * this->x) + (c * this->y);

  this->x = x_new; this->y = y_new;
}

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

```
int main( void )
{
  point_t pt;
  pt.x = 0;
  pt.y = 0;

  point_translate( &pt, 1, 0 );
  point_scale    ( &pt, 2 );
  return 0;
}
```



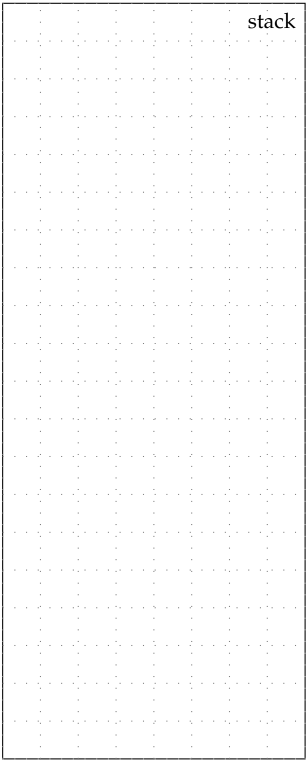
## 1.1. C++ Member Functions

- 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`

```
struct Point
{
  double x; // member fields
  double y; //

  // member functions

  void point_translate( double x_offset, double y_offset )
  {
    this->x += x_offset; this->y += y_offset;
  }

  void point_scale( double factor )
  {
    this->x *= factor; this->y *= factor;
  }

  void point_rotate( double angle )
  {
    ...
    double x_new = (c * this->x) - (s * this->y);
    double y_new = (s * this->x) + (c * this->y);
    this->x = x_new; this->y = y_new;
  }

  void point_print() const
  {
    std::printf("(%.2f,%.2f)", this->x, this->y );
  }
};
```

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

```
int main( void )
{
  Point pt;
  pt.x = 0;
  pt.y = 0;

  pt.point_translate( 1, 0 );
  pt.point_scale( 2 );
  return 0;
}
```



- Recall object-oriented prog in C

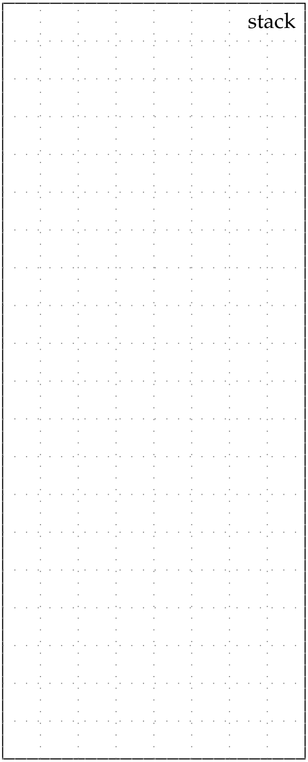
```
int main( void )
{
  point_t pt;
  pt.x = 0;
  pt.y = 0;

  point_translate( &pt, 1, 2 );
  point_scale    ( &pt, 2 );
  return 0;
}
```

```
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

```
struct Point
{
  double x; // member fields
  double y; //

  // member functions

  void translate( double x_offset, double y_offset )
  {
    x += x_offset; y += y_offset;
  }

  void scale( double factor )
  {
    x *= factor; y *= factor;
  }

  void rotate( double angle )
  {
    ...
    double x_new = (c * x) - (s * y);
    double y_new = (s * x) + (c * y);
    x = x_new; y = y_new;
  }

  void print() const
  {
    std::printf("(%.2f,%.2f)", x, y );
  }
};
```

```
int main( void )
{
  Point pt;
  pt.x = 0;
  pt.y = 0;

  pt.translate( 1, 2 );
  pt.scale( 2 );
  return 0;
}
```



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

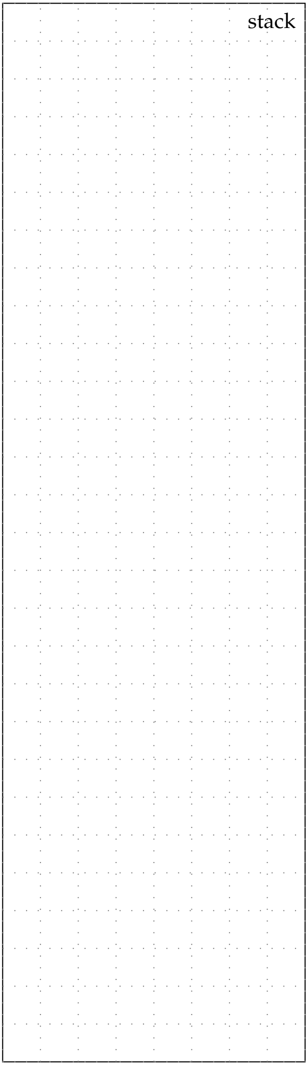
## 1.2. C++ Constructors

- 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

```
int main( void )
{
  Point pt;
  pt.construct();
  pt.translate( 1, 2 );
  pt.scale( 2 );
  return 0;
}
```

- 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

```
struct Point
{
  double x;
  double y;

  // default constructor
  Point()
  {
    x = 0.0;
    y = 0.0;
  }

  // non-default constructor
  Point( double x_, double y_ )
  {
    x = x_;
    y = y_;
  }

  ...
};

int main( void )
{
  Point pt0;
  Point pt1( 1, 2 );
  pt0 = pt1;
  pt0.translate( 1, 0 );
  return 0;
}
```



- Constructors automatically called with `new`

```
Point* pt0_p = new Point;     // constructor called
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

```
struct Point
{
  double x;
  double y;

  // default constructor
  Point()
  { x = 0.0; y = 0.0; }

  // non-default constructor
  Point( double x_, double y_ )
  { x = x_; y = y_; }

  ...
};
```

```
struct Point
{
  double x;
  double y;

  // default constructor
  Point()
   : x(0.0), y(0.0) { }

  // non-default constructor
  Point( double x_, double y_ )
   : x(x_), y(y_) { }

  ...
};
```

## 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)

```
Point operator+( const Point& pt0,
                 const Point& pt1 )
{
  Point tmp = pt0;
  tmp.translate( pt1.x, pt1.y );
  return tmp;
}

int main( void )
{
  Point ptA(1,2);
  Point ptB(3,4);
  Point ptC = ptA + ptB;
  return 0;
}
```

stack

stack

```
Point operator*( const Point& pt, double factor )
{
  Point tmp = pt;
  tmp.scale( factor );
  return tmp;
}

Point operator*( double factor, const Point& pt )
{
  Point tmp = pt;
  tmp.scale( factor );
  return tmp;
}

Point operator%( const Point& pt, double angle )
{
  Point tmp = pt;
  tmp.rotate( angle );
  return tmp;
}

Point operator%( double angle, const Point& pt )
{
  Point tmp = pt;
  tmp.rotate( angle );
  return tmp;
}
```

- Operator overloading enables elegant syntax for user-defined types

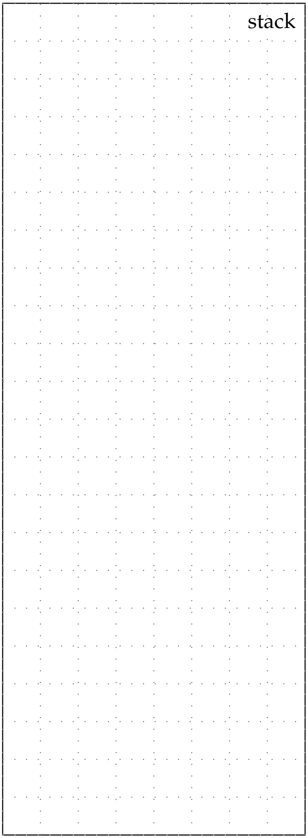
```
Point pt0(1,2);
pt0.translate(5,3);
pt0.rotate(45);
pt0.scale(1.5);
Point pt1 = pt0;
```

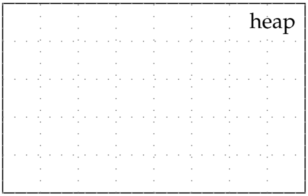
```
Point pt0(1,2);
Point pt1 = 1.5 * ( ( pt0 + Point(5,3) ) % 45 );
```

## 1.4. C++ Rule of Three

- What if point coordinates are allocated on the heap?

```
struct DPoint
{
  double* x_p;
  double* y_p;

  DPoint() {
    x_p = new double;
    y_p = new double;
    *x_p = 0.0;
    *y_p = 0.0;
  }

  void translate( double x_offset,
                  double y_offset )
  {
    *x_p += x_offset;
    *y_p += y_offset;
  }
  ...
};

int main( void )
{
  DPoint pt0;
  pt0.translate( 1, 0 );
  return 0;
}
```

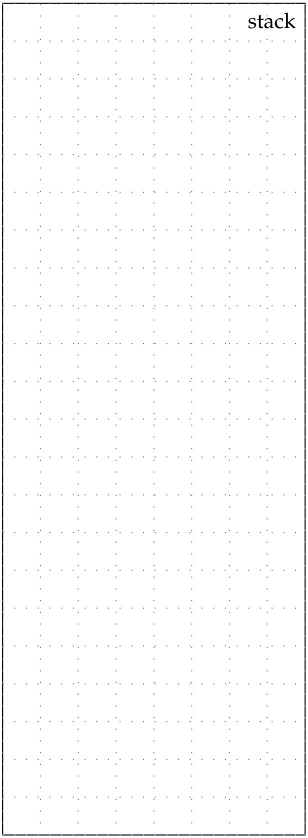


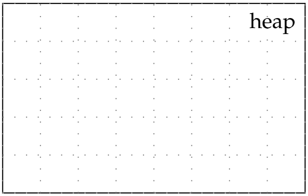


## C++ Destructors

- Destructors are special member functions for destroying an object

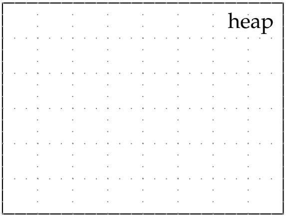
```
struct DPoint
{
  double* x_p;
  double* y_p;

  DPoint() {
    x_p = new double;
    y_p = new double;
    *x_p = 0.0;
    *y_p = 0.0;
  }

  ~DPoint() {
    delete x_p;
    delete y_p;
  }
  ...
};

int main( void )
{
  DPoint pt0;
  pt0.translate( 1, 0 );
  return 0;
}
```

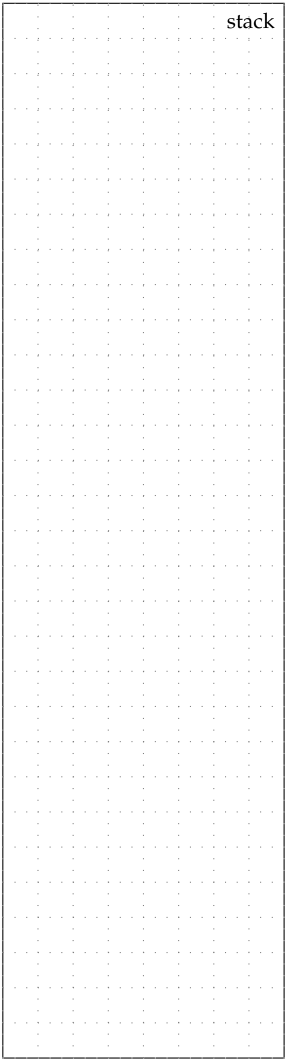




- What if we copy an object with dynamically allocated memory?

```
struct DPoint
{
  double* x_p;
  double* y_p;

  DPoint() {
    x_p = new double;
    y_p = new double;
    *x_p = 0.0;
    *y_p = 0.0;
  }

  ~DPoint() {
    delete x_p; delete y_p;
  }
  ...
};

int main( void )
{
  DPoint pt0;
  DPoint pt1 = pt0;
  pt0.translate( 1, 0 );
  return 0;
}
```

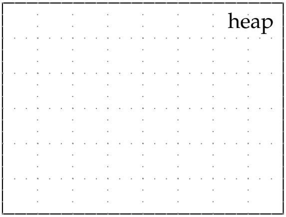


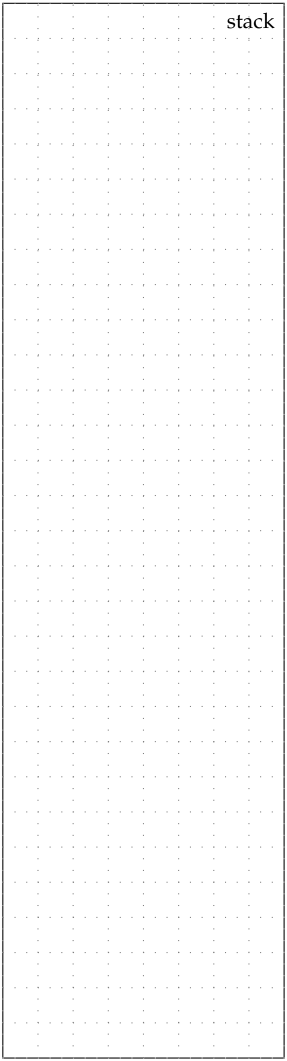


## C++ Copy Constructors

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

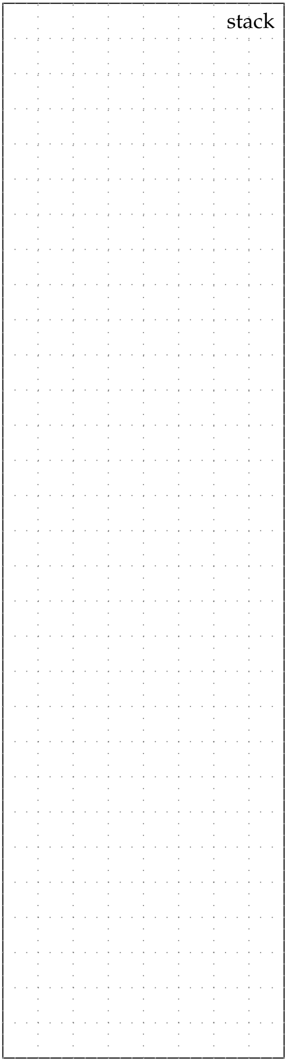
```
struct DPoint
{
  double* x_p;
  double* y_p;

  DPoint( const DPoint& pt ) {
    x_p = new double;
    y_p = new double;
    *x_p = *pt.x_p;
    *y_p = *pt.y_p;
  }

  ~DPoint() {
    delete x_p; delete y_p;
  }
  ...
};

int main( void )
{
  DPoint pt0;
  DPoint pt1 = pt0;
  pt0.translate( 1, 0 );
  return 0;
}
```

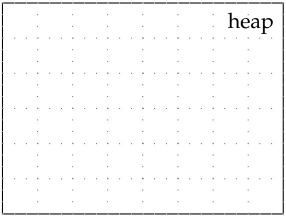




- What if we assign to an object with dynamically allocated memory?

```
struct DPoint
{
  double* x_p;
  double* y_p;

  DPoint() {
    x_p = new double;
    y_p = new double;
    *x_p = 0.0;
    *y_p = 0.0;
  }

  ~DPoint() {
    delete x_p; delete y_p;
  }
  ...
};

int main( void )
{
  DPoint pt0;
  DPoint pt1;
  pt1 = pt0;
  pt0.translate( 1, 0 );
  return 0;
}
```

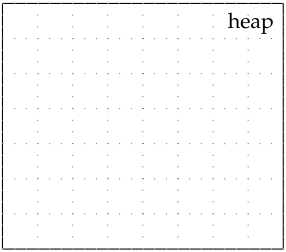


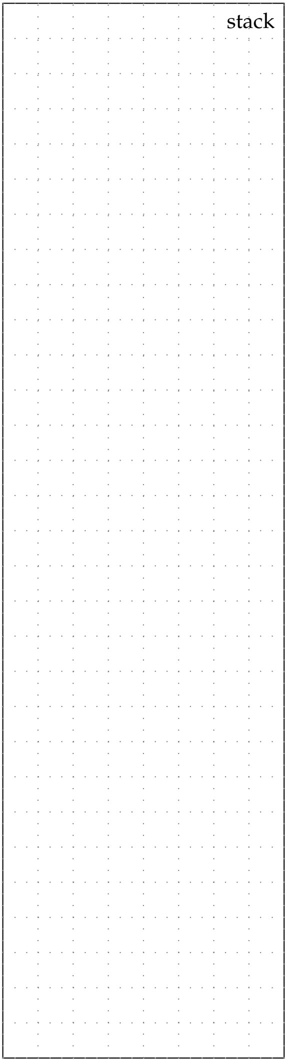


## C++ Assignment Operators

- An overloaded assignment operator will be called for assignment

```
struct DPoint
{
  double* x_p;
  double* y_p;

  DPoint&
  operator=( const DPoint& pt )
  {
    if ( this != &pt ) {
      *x_p = *pt.x_p;
      *y_p = *pt.y_p;
    }
    return *this;
  }
  ...
};

int main( void )
{
  DPoint pt0;
  DPoint pt1;
  pt1 = pt0;
  pt0.translate( 1, 0 );
  return 0;
}
```





### 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

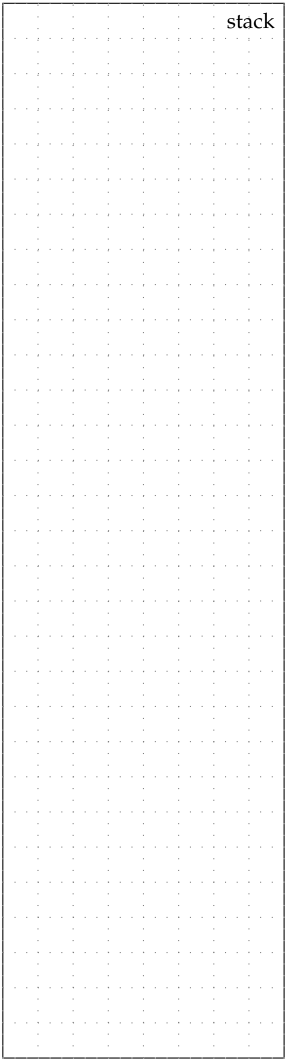
```
struct DPoint
{
  double* x_p;
  double* y_p;

  DPoint()
  {
    x_p = new double;
    y_p = new double;
    *x_p = 0.0;
    *y_p = 0.0;
  }

  DPoint( double x, double y )
  {
    x_p = new double;
    y_p = new double;
    *x_p = x;
    *y_p = y;
  }
```

```
  DPoint( const DPoint& pt )
  {
    x_p = new double;
    y_p = new double;
    *x_p = *pt.x_p;
    *y_p = *pt.y_p;
  }

  ~DPoint()
  {
    delete x_p;
    delete y_p;
  }

  DPoint&
  operator=( const DPoint& pt )
  {
    if ( this != &pt ) {
      *x_p = *pt.x_p;
      *y_p = *pt.y_p;
    }
    return *this;
  }

  ...
};
```

## C++ Exceptions and Destructors

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

```
struct DPoint
{
  double* x_p;
  double* y_p;

  ~DPoint() {
    delete x_p;
    delete y_p;
  }

  void translate( double x_offset,
                  double y_offset )
  {
    if (     (x_offset > 100)
          || (y_offset > 100) )
      throw 42;
    *x_p += x_offset;
    *y_p += y_offset;
  }

  ...
};

int main( void )
{
  try {
    DPoint pt0;
    pt0.translate( 1e9, 0 );
  }
  catch ( int e ) {
    return e;
  }
  return 0;
}
```



## 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

```
char* get_first_word( const char* str )
{
  // Determine how many characters are in first word
  int len = 0;
  while ( (str[len] != ' ') && (str[len] != '\0') )
    len++;

  // Dynamically allocate space for first word
  char* first_word = new char[len+1];

  // Copy the first word to the new allocated string
  for ( int i = 0; i < len; i++ )
    first_word[i] = str[i];
  first_word[len] = '\0';

  return first_word;
}

int main( void )
{
  char* x = get_first_word( "foo bar" );
  char* y = get_first_word( "foo" );
  char* z = get_first_word( "foo bar baz" );
  return 0;
};
```

```
struct String
{
  String()                          { m_str = nullptr;        }
  String( int size )                { m_str = new char[size]; }
  char& operator[]( int idx ) { return m_str[idx];      }
  ~String()                         { delete[] m_str;         }
  String( const String& str ) { copy( str.m_str );      }
  const char* c_str() const   { return m_str;           }

  String& operator=( const String& str )
  {
    if ( this != &str ) {
      delete[] m_str;
      copy( str.m_str );
    }
    return *this;
  }

  void copy( const char* s )
  {
    // Determine how many characters are in given string
    int len = 0;
    while ( s[len] != '\0' )
      len++;

    // Dynamically allocate space for copy
    m_str = new char[len+1];

    // Copy the given string to the newly allocated string
    for ( int i = 0; i < len+1; i++ )
      m_str[i] = s[i];
  }

  char* m_str;
};
```

```
String get_first_word( const char* str )
{
  int len = 0;
  while ( (str[len] != ' ') && (str[len] != '\0') )
    len++;

  String first_word(len+1);

  for ( int i = 0; i < len; i++ )
    first_word[i] = str[i];

  first_word[len] = '\0';
  return first_word;
}

int main( void )
{
  String x = get_first_word( "foo bar" );
  String y = get_first_word( "foo" );
  String z = get_first_word( "foo bar baz" );
  return 0;
};
```

- 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

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

- 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

```
struct Point
{
 private:
  double x; double y;

 public:
  // default constructor
  Point() { x = 0.0; y = 0.0; }

  // non-default constructor
  Point( double x_, double y_ ) { x = x_; y = y_; }
  ...
};
```

- 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

```
class Point // almost always use class instead of struct
{
 public:    // always explicitly use public ...
 private:   // ... or private
};
```

- 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

```
int main( void )
{
  Point pt0(1,2);
  Point pt1 = pt0;
  pt1.translate(2.0,2.0);

  std::printf("initial point = ");
  pt0.print();
  std::printf("\n");
  std::printf("translate by %.2f,%.2f\n", 2.0, 2.0 );
  std::printf("new point = ");
  pt1.print();
  std::printf("\n");
}
```

- Clever use of operator overloading can provide cleaner syntax

```
struct ostream
{
  // internal stream state
};

ostream cout;

ostream& operator<<( ostream& os, int i )
{
  std::printf("%d",i); return os;
}

ostream& operator<<( ostream& os, double d )
{
  std::printf("%.2f",d); return os;
}

ostream& operator<<( ostream& os, const char* str )
{
  std::printf("%s",str); return os;
}

ostream& operator<<( ostream& os, const Point& pt )
{
  pt.print(); return os;
}

struct EndOfLine
{ };

EndOfLine endl;

ostream& operator<<( ostream& os, const EndOfLine& endl )
{
  std::printf("\n"); return os;
}
```

```
int main( void )
{
  Point pt0(1,2);
  Point pt1 = pt0;
  pt1.translate(2.0,2.0);

  cout << "initial point = " << pt0 << endl;
  cout << "translate by " << 2.0 << "," << 2.0 << endl;
  cout << "new point = " << pt1 << endl;
}
```

- 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

```
typedef struct
{
  // implementation specific
}
slist_int_t;

void slist_int_construct  ( slist_int_t* this );
void slist_int_destruct   ( slist_int_t* this );
void slist_int_push_front ( slist_int_t* this, int v );
void slist_int_reverse    ( slist_int_t* this );
```

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

```
class SListInt
{
 public:
  SListInt();                   // constructor
  ~SListInt();                  // destructor
  void push_front( int v );     // member function
  void reverse();               // member function

  // implementation specific
};
```

- 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

```
typedef struct _slist_int_node_t
{
  int                        value;
  struct _slist_int_node_t* next_p;
}
slist_int_node_t;

typedef struct
{
  slist_int_node_t* head_p;
}
list_int_t;
```

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

```
class SListInt
{
 public:
  SListInt();                  // constructor
  ~SListInt();                 // destructor
  void push_front( int v );    // member function
  void reverse();              // member function

  struct Node                  // nested struct declaration
  {
    int   value;
    Node* next_p;
  };

  Node* m_head_p;              // member field
};
```

- Implementation for a C singly linked list data structure

```c
void slist_int_construct(
  slist_int_t* this )
{
  this->head_p = NULL;
}

void slist_int_push_front(
  slist_int_t* this, int v )
{
  slist_int_node_t* new_node_p
   = malloc(sizeof(slist_int_node_t));
  new_node_p->value  = v;
  new_node_p->next_p = this->head_p;
  this->head_p       = new_node_p;
}

void slist_int_destruct(
  slist_int_t* this )
{
  while ( this->head_p != NULL ) {
    list_int_node_t* temp_p
      = this->head_p->next_p;
    free( this->head_p );
    this->head_p = temp_p;
  }
}
```

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

```cpp
SListInt::SListInt()

{
  m_head_p = nullptr;
}

void SListInt::push_front( int v )

{
  Node* new_node_p
    = new Node;
  new_node_p->value  = v;
  new_node_p->next_p = m_head_p;
  m_head_p           = new_node_p;
}

SListInt::~SListInt()

{
  while ( m_head_p != nullptr ) {
    Node* temp_p
      = m_head_p->next_p;
    delete m_head_p;
    m_head_p = temp_p;
  }
}
```

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

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

void SListInt::push_front( int v )
{
  Node* new_node_p
    = new Node;
  new_node_p->value = v;
  new_node_p->next_p = m_head_p;
  m_head_p = new_node_p;
}

int main( void )
{
  SListInt lst;
  lst.push_front(12);
  lst.push_front(11);
  lst.push_front(10);

  SListInt::Node* node_p
    = lst.m_head_p;
  while ( node_p != nullptr ) {
    int value = node_p->value;
    node_p = node_p->next_p;
  }

  return 0;
}
```

stack

heap

## 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

```
class SListInt
{
  ...

 private:

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

  Node* m_head_p;

 public:

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

   private:
    Node* m_node_p;
  };

  Itr begin();
  Itr end();

};
```

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

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

int& SListInt::Itr::get()
{
  assert( m_node_p != nullptr );
  return m_node_p->value;
}

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

SListInt::Itr SListInt::begin()
{
  return Itr(m_head_p);
}

SListInt::Itr SListInt::end()
{
  return Itr(nullptr);
}
```

```
SListInt::Node* node_p
  = list.m_head_p;
while ( node_p != nullptr ) {
  int value = node_p->value
  printf( "%d\n", value );
  node_p = node_p->next_p;
}
```

```
SListInt::Itr itr
  = list.begin();
while ( !itr.eq(list.end()) ) {
  int value = itr.get();
  printf( "%d\n", value );
  itr.next();
}
```

- We can use operator overloading to improve iterator syntax

```
// postfix increment operator (itr++)
SListInt::Itr operator++( SListInt::Itr& itr, int )
{
  SListInt::Itr itr_tmp = itr; itr.next(); return itr_tmp;
}

// prefix increment operator (++itr)
SListInt::Itr& operator++( SListInt::Itr& itr )
{
  itr.next(); return itr;
}

// dereference operator (*itr)
int& operator*( SListInt::Itr& itr )
{
  return itr.get();
}

// not-equal operator (itr0 != itr1)
bool operator!=( const SListInt::Itr& itr0,
                 const SListInt::Itr& itr1  )
{
  return !itr0.eq( itr1 );
}
```

```
SListInt::Itr itr = lst.begin();
while ( !itr.eq(lst.end()) ) {
  int value = itr.get();
  printf( "%d\n", value );
  itr.next();
}
```

```
SListInt::Itr itr = lst.begin();
while ( itr != lst.end() ) {
  int value = *itr;
  printf( "%d\n", value );
  ++itr;
}
```

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

- 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

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

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

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

# 3. C++ Inheritance

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

  int get_pts() const
  {
    return 1;
  }

  char get_col() const
  {
   return m_col;
  }

  int get_row() const
  {
   return m_row;
  }

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

   m_col = col;
   m_row = row;
  }

  void print() const
  {
    std::printf( "pawn@%c%d",
                 m_col, m_row );
  }

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

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

  int get_pts() const
  {
    return 5;
  }

  char get_col() const
  {
   return m_col;
  }

  int get_row() const
  {
   return m_row;
  }

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

   m_col = col;
   m_row = row;
  }

  void print() const
  {
    std::printf( "rook@%c%d",
                 m_col, m_row );
  }

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

- Object-oriented design without inheritance

- We want to be able to ...

... create algorithms and data structures that work for all pieces
... easily reuse implementation code across pieces

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

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

## 3.1. C++ Implementation Inheritance

- Object-oriented design with implementation inheritance

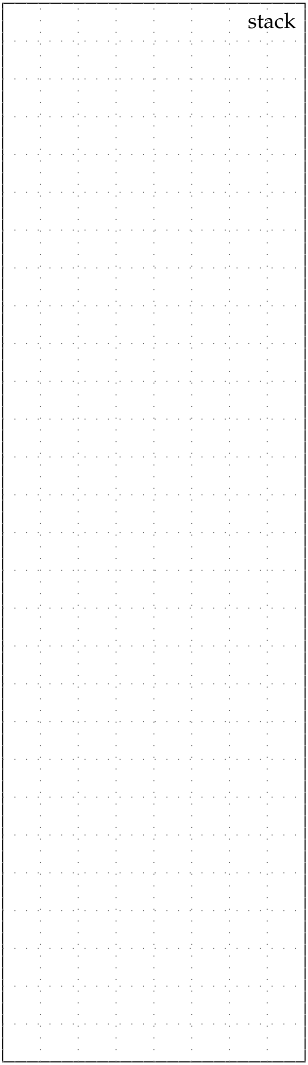
```cpp
class Piece
{
 public:

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

  char get_col() const
  {
   return m_col;
  }

  int get_row() const
  {
   return m_row;
  }






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







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

  char m_col;
  int  m_row;

};
```

```cpp
class Rook : public Piece
{
 public:

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











  int get_pts() const
  {
    return 5;
  }

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

   Piece::move( col, row );
  }

  void print() const
  {
    std::printf( "rook@%c%d",
                 m_col, m_row );
  }






};
```

```cpp
class Piece
{
  Piece( char col, int row )
    : m_col(col), m_row(row)
  { }

  char get_col() const
  {
    return m_col;
  }

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

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

  int get_pts() const
  {
    return 1;
  }
  ...
};

int main( void )
{
  Pawn p('a',2);
  char col = p.get_col();
  return 0;
}
```

stack

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

  char get_col() const
  {
    return m_col;
  }

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

class Pawn : public Piece
{
  int get_pts() const { return 1; }
  ...
};

class Rook : public Piece
{
  int get_pts() const { return 5; }
  ...
};

int main( void )
{
  Pawn p('a',2);
  Rook r('h',3);
  int pts0 = p.get_pts();
  int pts1 = r.get_pts();
  int sum  = pts0 + pts1;
  return 0;
}
```



## 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

```
Rook rook('h',3);

// implicit type conversion
Piece* piece_p = &rook;

// can only call methods defined in Piece
piece_p->get_col();

// explicit type casting to correct type
Rook* rook_p = (Rook*) piece_p;

// can call methods defined in Piece or Rook
rook_p->get_pts();

// explicit type casting to wrong type (segfault)
Pawn* pawn_p = (Pawn*) piece_p;

// ptr == nullptr if piece_p is not pointer to a Rook
Rook* ptr = dynamic_cast<Rook*>(piece_p);
```

## Dynamic Polymorphism

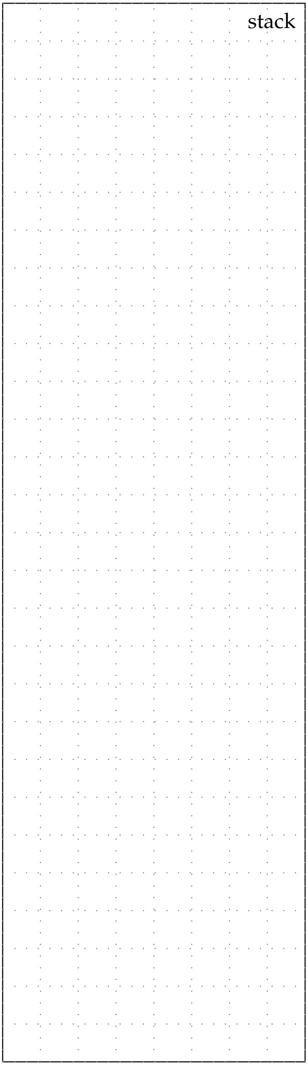
- Consider the following code snippet from previous example

```
int pts0 = p.get_pts();
int pts1 = r.get_pts();
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

```
int calc_pts( Piece* p0, Piece* p1 )
{
  int pts0 = p0->get_pts();
  int pts1 = p1->get_pts();
  return pts0 + pts1;
}
```

```
class Piece
{
  Piece( char col, int row )      { ...
  char get_col() const            { ...
  int  get_row() const            { ...
  void move( char col, int row ) { ...
  ...
};

class Pawn : public Piece
{
  int get_pts() const { return 1; }
  ...
};

class Rook : public Piece
{
  int get_pts() const { return 5; }
  ...
};

int calc_pts( Piece* p0, Piece* p1 )
{
  int pts0 = p0->get_pts();
  int pts1 = p1->get_pts();
  return pts0 + pts1;
}

int main( void )
{
  Pawn p('a',2);
  Rook r('h',3);
  int sum = calc_pts( &p, &r );
  return 0;
}
```

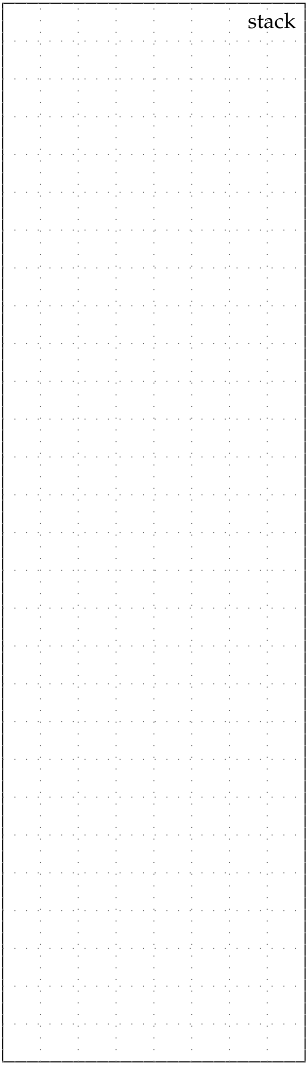


```
// enum for all possible
// implementation types
enum PieceType { PAWN, ROOK };

class Piece
{
 public:

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

  PieceType get_type() const
  {
    return m_type;
  }

  char get_col() const
  {
    return m_col;
  }

  int get_row() const
  {
    return m_row;
  }

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

 protected:

  // explicit type field
  PieceType m_type;

  char      m_col;
  int       m_row;

};
```

- 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

```
class Rook : public Piece
{
 public:

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

  int get_pts() const
  {
    return 5;
  }

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

   Piece::move( col, row );
  }

  void print() const
  {
    std::printf( "rook@%c%d",
                 m_col, m_row );
  }
};
```

```
class Pawn : public Piece
{
  int get_pts() const { return 1; }
  ...
};

int calc_pts( Piece* p0, Piece* p1 )
{
  int pts0;
  if ( p0->get_type() == PAWN ) {
    Pawn* pawn_p = (Pawn*) p0;
    pts0 = pawn_p->get_pts();
  }
  else if ( p0->get_type() == ROOK ) {
    Rook* rook_p = (Rook*) p0;
    pts0 = rook_p->get_pts();
  }

  int pts1;
  if ( p1->get_type() == PAWN ) {
    Pawn* pawn_p = (Pawn*) p1;
    pts1 = pawn_p->get_pts();
  }
  else if ( p1->get_type() == ROOK ) {
    Rook* rook_p = (Rook*) p1;
    pts1 = rook_p->get_pts();
  }

  return pts0 + pts1;
}

int main( void )
{
  Pawn p('a',2);
  Rook r('h',3);
  int sum = calc_pts( &p, &r );
  return 0;
}
```

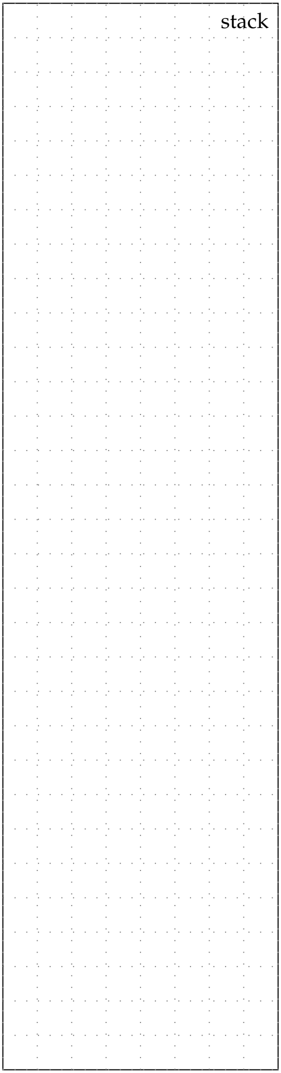


```
enum PieceType { PAWN, ROOK };

class Piece
{
 public:

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

  int get_pts() const
  {
    if ( m_type == PAWN ) {
      Pawn* pawn_p = (Pawn*) this;
      return pawn_p->get_pts();
    }
    else if ( m_type == ROOK ) {
      Rook* rook_p = (Rook*) this;
      return rook_p->get_pts();
    }
  }

  char get_col() const
  {
    return m_col;
  }

  int get_row() const
  {
    return m_row;
  }

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

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

- 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

```
class Rook : public Piece
{
 public:

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

  int get_pts() const
  {
    return 5;
  }

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

   Piece::move( col, row );
  }

  void print() const
  {
    std::printf( "rook@%c%d",
                 m_col, m_row );
  }
};
```

```cpp
class Piece
{
  ...
  int get_pts() const
  {
    if ( m_type == PAWN ) {
      Pawn* pawn_p = (Pawn*) this;
      return pawn_p->get_pts();
    }
    else if ( m_type == ROOK ) {
      Rook* rook_p = (Rook*) this;
      return rook_p->get_pts();
    }
  }
};

class Pawn : public Piece
{
  int get_pts() const { return 1; }
  ...
};

int calc_pts( Piece* p0, Piece* p1 )
{
  int pts0 = p0->get_pts();
  int pts1 = p1->get_pts();
  return pts0 + pts1;
}

int main( void )
{
  Pawn p('a',2);
  Rook r('h',3);
  int sum = calc_pts( &p, &r );
  return 0;
}
```



## 3.3. C++ Interface Inheritance

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

```
class Piece
{
 public:

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

  // pure virtual function
  virtual int get_pts() const = 0;

  char get_col() const
  {
    return m_col;
  }

  int get_row() const
  {
    return m_row;
  }

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

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

```
class Rook : public Piece
{
 public:

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

  int get_pts() const
  {
    return 5;
  }

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

   Piece::move( col, row );
  }

  void print() const
  {
    std::printf( "rook@%c%d",
                 m_col, m_row );
  }
};
```

```
class Piece
{
  virtual int get_pts() const = 0;
  ...
};

class Pawn : public Piece
{
  int get_pts() const { return 1; }
  ...
};

class Rook : public Piece
{
  int get_pts() const { return 5; }
  ...
};

int calc_pts( Piece* p0, Piece* p1 )
{
  int pts0 = p0->get_pts();
  int pts1 = p1->get_pts();
  return pts0 + pts1;
}

int main( void )
{
  Pawn p('a',2);
  Rook r('h',3);
  int sum = calc_pts( &p, &r );
  return 0;
}
```

stack

## Abstract base classes

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

```
class IPiece
{
 public:
  virtual ~IPiece() { }
  virtual int  get_pts() const = 0;
  virtual char get_col() const = 0;
  virtual int  get_row() const = 0;
  virtual void move( char col, int row ) = 0;
  virtual void print() const  = 0;
};

class Rook : public IPiece
{
 public:
  Rook( char col, int row ) : m_col(col), m_row(row) { }
  ~Rook() { }

  int  get_pts() const { return 5;     }
  char get_col() const { return m_col; }
  int  get_row() const { return m_row; }

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

  void print() const
  {
    std::printf( "rook@%c%d", m_col, m_row );
  }

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

## 3.4. Revisiting Composition vs. Generalization

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

  char get_col() const
  {
    return m_col;
  }

  int get_row() const
  {
    return m_row;
  }

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

  void print() const
  {
    std::printf( "%c%d",
                 m_col, m_row );
  }

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

```cpp
class Pawn : public IPiece
{
 public:

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

  ~Pawn()
  { }

  int get_pts() const
  {
    return 1;
  }

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

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

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

  void print() const
  {
    std::printf("pawn@");
    m_pos.print();
  }

 private:
  Position m_pos;
};
```

# 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

```
class IObject
{
 public:

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

};

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

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

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

```
class Integer : public IObject
{
 public:
  Integer()           : m_data(0)        { }
  Integer( int data ) : m_data(data)     { }

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

  bool eq( const IObject& rhs ) const
  {
    const Integer* rhs_p = dynamic_cast<const Integer*>( &rhs );
    if ( rhs_p == nullptr )
      return false;
    else
      return ( m_data == rhs_p->m_data );
  }

  bool lt( const IObject& rhs ) const
  {
    const Integer* rhs_p = dynamic_cast<const Integer*>( &rhs );
    if ( rhs_p == nullptr )
      return false;
    else
      return ( m_data < rhs_p->m_data );
  }

  void print() const
  {
    std::printf( "%d", m_data );
  }

 private:
  int m_data;
};
```

```
class IObject
{
 public:
   virtual bool
   eq( const IObject& rhs ) const = 0;
  ...
};

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

class Integer : public IObject
{
 public:
  Integer( int data ) : m_data(data) { }

  bool eq( const IObject& rhs ) const {
   const Integer* rhs_p
    = dynamic_cast<const Integer*>(&rhs);
   if ( rhs_p == nullptr )
     return false;
   else
     return (m_data == rhs_p->m_data);
  }
  ...
 private:
  int m_data;
};

int main( void )
{
  Integer a(2);
  Integer b(3);
  bool c = ( a == b );
  return 0;
}
```

stack

## 4.1. Singly Linked List Interface

- Object-oriented list which stores `int`s

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

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

   private:
    Node* m_node_p;
  };

  Itr begin();
  Itr end();

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

  Node* m_head_p;
};
```

- Object-oriented list which stores `IObject`s

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

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

   private:
    Node* m_node_p;
  };

  Itr begin();
  Itr end();

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

  Node* m_head_p;
};
```

## 4.2. Singly Linked List Implementation

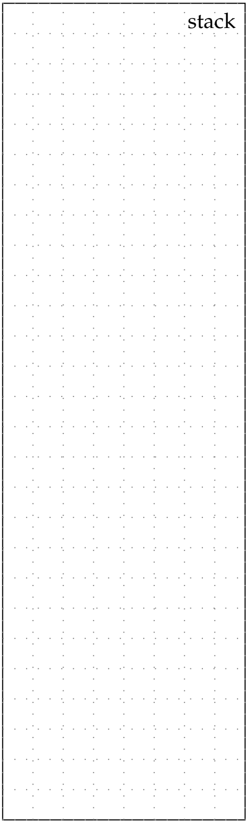
- Implementation for singly linked list to store `ints`

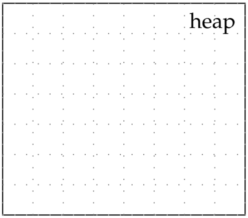
- Implementation for singly linked list to store `IObjects`

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

void SListInt::push_front( int v )

{
  Node* new_node_p    = new Node;
  new_node_p->value   = v;
  new_node_p->next_p = m_head_p;
  m_head_p            = new_node_p;
}

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

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

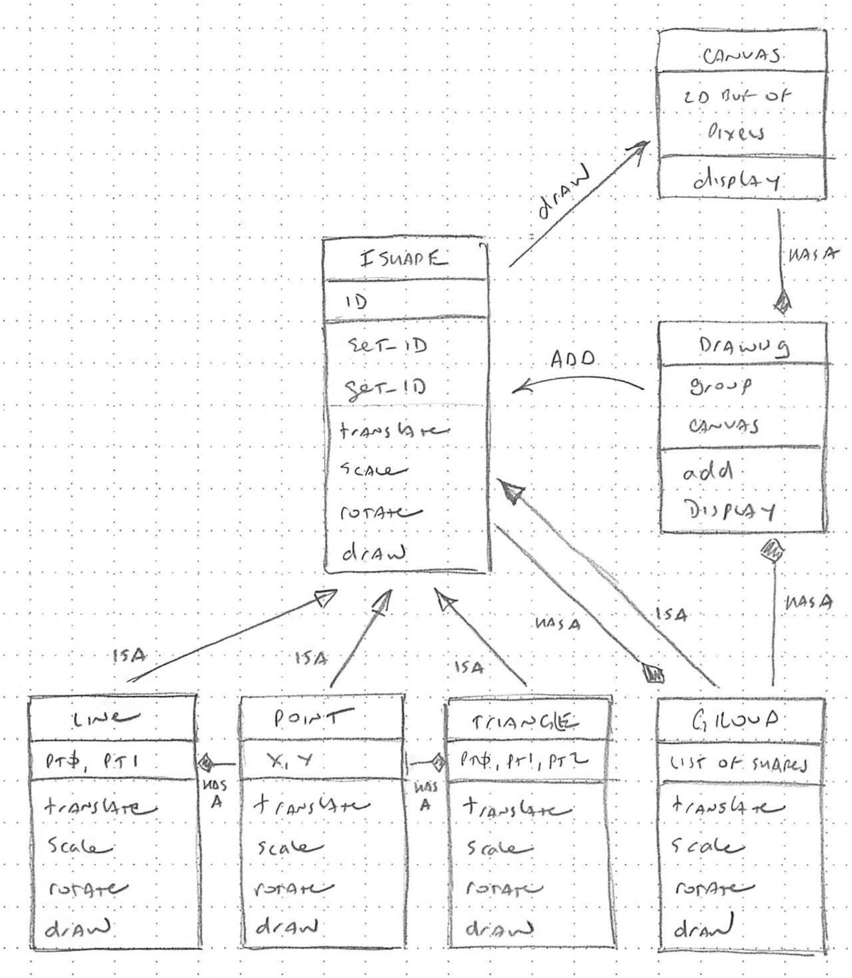
```
SListIObj::SListIObj()
{
  m_head_p = nullptr;
}

void SListIObj::push_front(
                const IObject& v )
{
  Node* new_node_p    = new Node;
  new_node_p->obj_p   = v.clone();
  new_node_p->next_p = m_head_p;
  m_head_p            = new_node_p;
}

SListIObj::~SListIObj()
{
  while ( m_head_p != nullptr ) {
    Node* temp_p = m_head_p->next_p;
    delete m_head_p->obj_p;
    delete m_head_p;
    m_head_p = temp_p;
  }
}
```

```
class IObject
{
 public:
   virtual IObject* clone( ) const = 0;
  ...
};

class Integer : public IObject
{
 public:
  Integer( int data ) : m_data(data) { }
  Integer* clone() const {
    return new Integer( *this );
  }
  ...
 private:
  int m_data;
};

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

void SListIObj::push_front(
                const IObject& v ) {
  Node* new_node_p   = new Node;
  new_node_p->obj_p  = v.clone();
  new_node_p->next_p = m_head_p;
  m_head_p           = new_node_p;
}

int main( void )
{
  SListIObj list;
  Integer a(12);
  list.push_front(a);
  return 0;
}
```





# 5. Drawing Framework Case Study



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