ECE 2400 Computer Systems Programming
Fall 2017

Topic 13: C++ Object-Oriented Programming

School of Electrical and Computer Engineering
Cornell University

revision: 2017-11-02-21-26

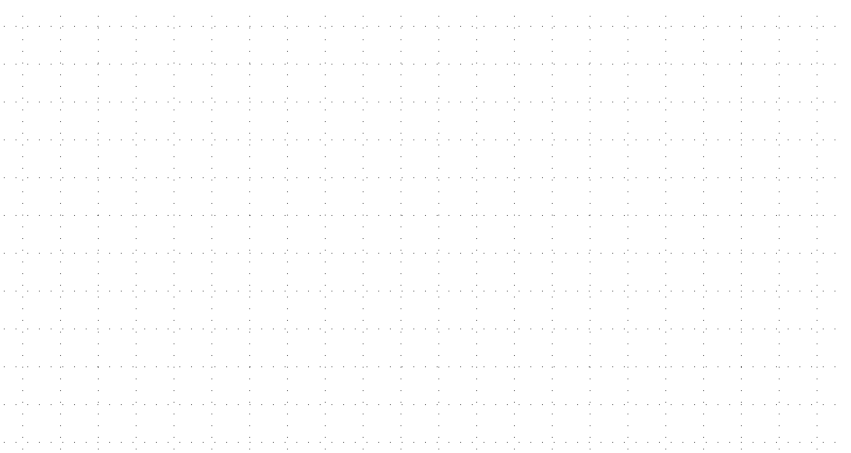
Procedural programming

- Programming model organized around procedures
- Procedures take input data, process it, produce output data
- Focus is on the logic to process data

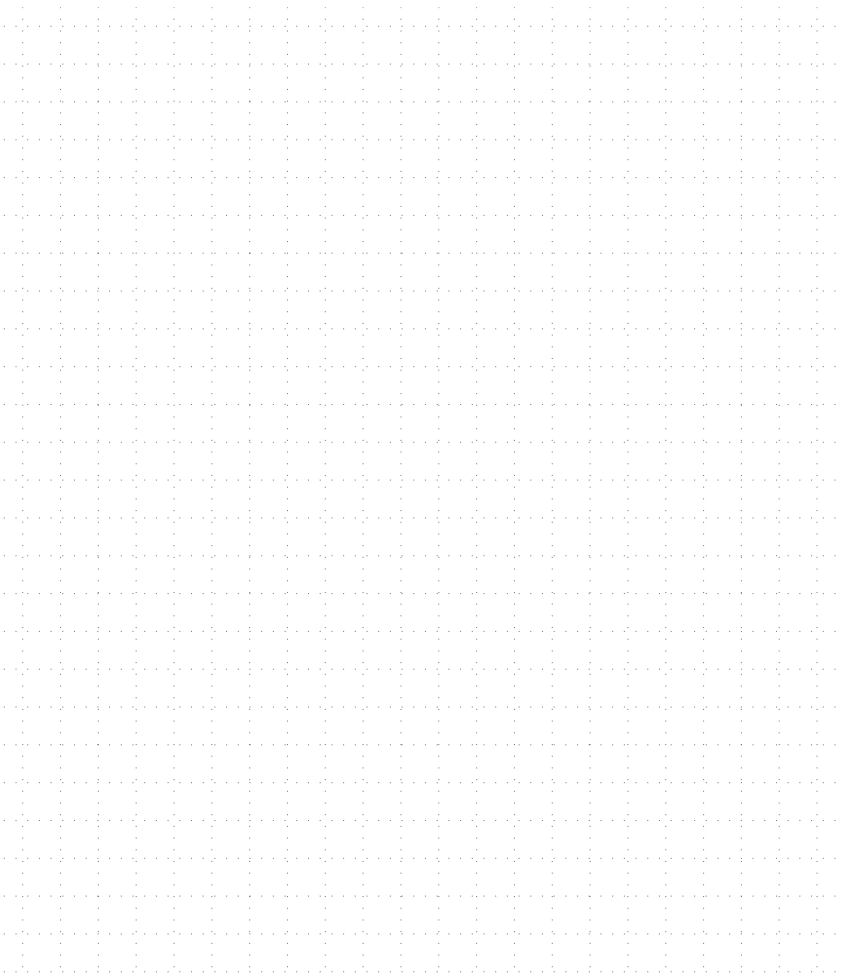
Object-oriented programming

- Programming model organized around objects
- Objects contain data and actions to perform on data
- 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



Pseudocode for shapes and drawings

Adding shapes to a drawing and displaying on screen

```
set drawing to new Drawing
add point0    to drawing
add line0     to drawing
add triangle0 to drawing

set group to new Group
add triangle1 to group
add triangle2 to group
add group     to drawing

display drawing on screen
```

Pseudocode for display drawing

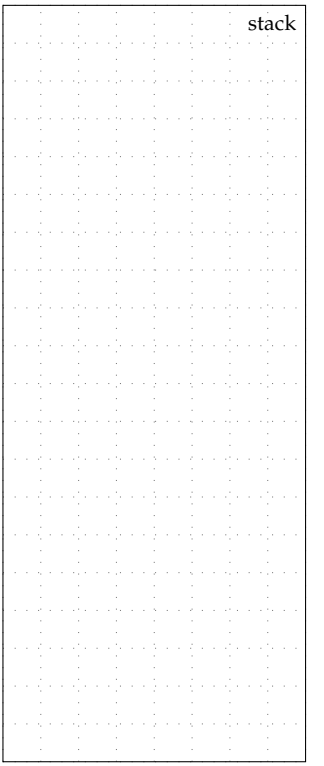
```
set canvas to new Canvas
for shape in drawing's internal group
  draw shape on canvas
display canvas on screen
```

1. Points, Lines, Triangles

- Perfectly possible to use object-oriented programming in C

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

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

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

void point_rotate( point_t* point_p, 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 * point_p->x) - (s * point_p->y);
  double y_new = (s * point_p->x) + (c * point_p->y);

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

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

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

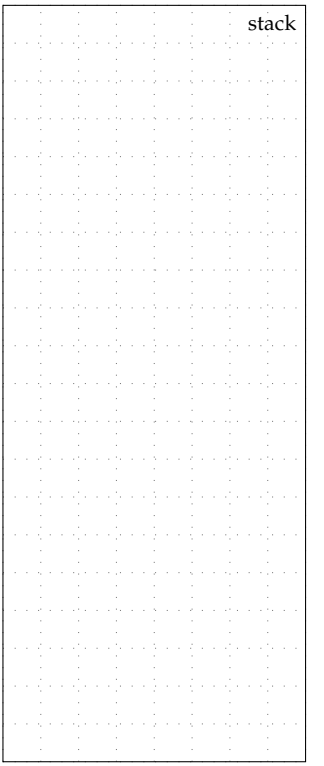


1.1. C++ Member Functions

- C++ allows functions to be defined *within* the `struct` namespace
- C++ `struct` has both member fields and member functions

```
1  struct Point
2  {
3    double x; // member fields
4    double y; //
5
6    // (static) member functions
7
8    static void translate( Point* point_p
9                           double x_offset, double y_offset )
10   {
11     point_p->x += x_offset; point_p->y += y_offset;
12   }
13
14   static void scale( Point* point_p double scale )
15   {
16     point_p->x *= scale; point_p->y *= scale;
17   }
18
19   static void rotate( Point* point_p double angle )
20   {
21     ...
22     double x_new = (c * point_p->x) - (s * point_p->y);
23     double y_new = (s * point_p->x) + (c * point_p->y);
24     point_p->x = x_new; point_p->y = y_new;
25   }
26
27   static void print( Point* point )
28   {
29     std::printf("(%.2f,%.2f)", point_p->x, point_p->y );
30   }
31 };
```

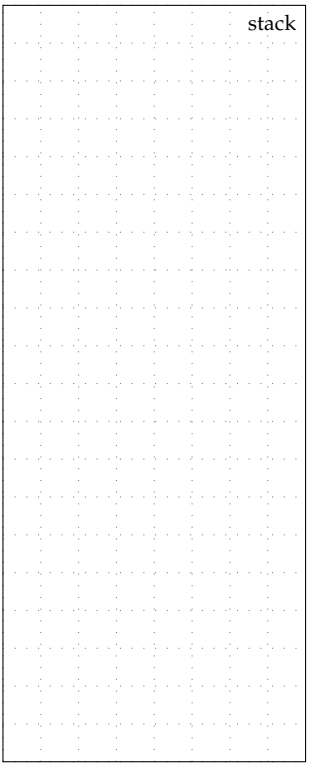
```
1  int main( void )
2  {
3    Point pt;
4    pt.x = 1;
5    pt.y = 2;
6
7    Point::translate( &pt, 1, 0 );
8    Point::scale( &pt, 2 );
9    return 0;
10 }
```



- Non-static member functions have an implicit `this` pointer
- The `this` pointer serves same purpose as `point_p`
- Non-static member functions which do not modify fields are `const`
- Non-static member functions are accessed using the dot (`.`) operator in the same way we access fields

```
1  struct Point
2  {
3    double x; // member fields
4    double y; //
5
6    // (non-static) member functions
7
8    void translate( double x_offset, double y_offset )
9    {
10     this->x += x_offset; this->y += y_offset;
11   }
12
13   void scale( double scale )
14   {
15     this->x *= scale; this->y *= scale;
16   }
17
18   void 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 print() const
27   {
28     std::printf("(%.2f,%.2f)", this->x, this->y );
29   }
30 };
```

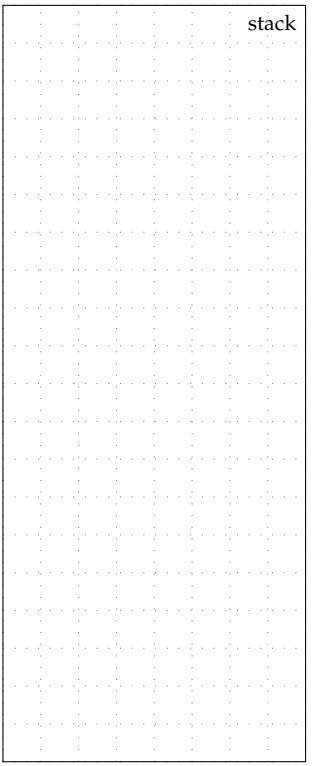
```
1  int main( void )
2  {
3    Point pt;
4    pt.x = 1;
5    pt.y = 2;
6
7    pt.translate( 1, 0 );
8    pt.scale( 2 );
9    return 0;
10 }
```



- Member fields are in scope within every non-static member function
- No need to explicitly use `this` pointer

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

  // (non-static) member functions

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

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

  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 );
  }
};
```

```
1  int main( void )
2  {
3    Point pt;
4    pt.x = 1;
5    pt.y = 2;
6
7    pt.translate( 1, 0 );
8    pt.scale( 2 );
9    return 0;
10 }
```



- Static member fields/functions are associated with the struct
- Non-static member fields/functions are associated with the object
- An object is just an instance of a struct with member functions

1.2. C++ Constructors and Destructors

- How do we construct and destruct an object?
- In C, we used `foo_construct` and `foo_destruct`
- In C++, we could add `construct` and `destruct` member functions

```
1  int main( void )
2  {
3    Point pt;
4    pt.construct();
5    pt.x = 1;
6    pt.y = 2;
7
8    pt.translate( 1, 0 );
9    pt.scale( 2 );
10   pt.destruct();
11   return 0;
12 }
```

- What if we call `translate` before `construct`?
- What if we call `translate` after `destruct`?
- What if `translate` throws an exception?

- C++ adds support for language-level constructors and destructors
- Involves new syntax and semantics

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

  // constructor

  Point()
  {
    x = 0.0;
    y = 0.0;
  }

  // destructor

  ~Point()
  { }

  ...
};
```

```cpp
int main( void )
{
  Point pt0; // constructor called
  pt0.x = 1;
  pt0.y = 2;

  try {
    Point pt1; // constructor called
    pt1.x = 1;
    pt1.y = 2;

    // destructor called if
    // exception is thrown
    throw "example exception";

    pt1.translate( 0, 1 );
    pt1.scale( 0.5 );
  }
  catch ( const char* e ) {
    std::printf( "ERROR: %s\n", e );
  }

  pt0.translate( 1, 0 );
  pt0.scale( 2 );
  return 0; // destructor called
}
```

http://cpp.sh/8j5w

- Constructors automatically called with `new`
- Destructors automatically called with `delete`

```cpp
  // always use () when allocating one object
  Point* pt0_p = new Point();   // constructor called
  delete pt0_p;                 // destructor called

  Point* pt1_p = new Point[4];  // constructor called 4 times
  delete[] pt1_p;               // destructor called  4 times
```

- Constructors can take arguments to initialize members
- Copy constructors used for copying object
  - Used in initialization statements
  - Used when object is passed by value to a function
- 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( int x_, int y_ )
  { x = x_; y = y_; }

  // copy constructor
  Point( const Point& pt )
  { x = pt.x; y = pt.y; }

  ...
};
```

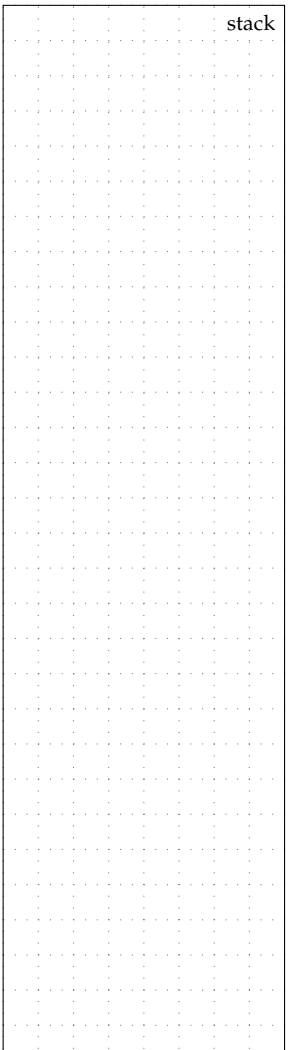
```
struct Point
{
  double x; double y;

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

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

  // copy constructor
  Point( const Point& pt )
   : x(pt.x), y(pt.y) { }

  ...
};
```

```
int main( void )
{
  Point pt0;          // default constructor called
  Point pt1( 1, 2 );  // non-default constructor called
  Point pt2 = pt1;    // copy constructor called
  return 0;
}
```

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 an overloaded function (either a member function or a free function)

```
1  Point
2  operator+( const Point& pt0,
3             const Point& pt1 )
4  {
5    // calls copy constructor
6    Point tmp = pt0;
7    tmp.translate( pt1.x, pt1.y );
8    return tmp;
9  }
10
11 int main( void )
12 {
13   Point pt0(1,2);
14   Point pt1(3,4);
15   Point pt2 = pt0 + pt1;
16   return 0;
17 }
```



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

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

- Overloading the assignment (=) operator is also possible
- Important to correctly delete dynamically allocated memory in the LHS of the assignment operator, since we are essentially overwriting the object on the LHS

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

  // copy constructor
  Point( const Point& pt )
  {
    // Copy data from pt to newly allocated memory
    x = pt.x; y = pt.y;
  }

  // destructor
  ~Point()
  {
    // Free any dynamically allocated memory
  }

  // overload assignment operator
  Point& operator=( const Point& pt )
  {
    // Free any dynamically allocated memory
    // Copy data from pt to newly allocated memory
    x = pt.x; y = pt.y;
  }

  ...
};
```

- Classes included an implicitly defined copy constructor, destructor, and assignment operator if they are not specified
- Implicitly definitions perform a shallow copy
- Rule of Three: If you define one (e.g., to do a deep copy) you should probably define all three appropriately

1.4. Classes for Lines and Triangles

```
struct Line
{
  Point pt0;
  Point pt1;

  Line()
  { }

  Line( Point pt0_, Point pt1_ )
    : pt0(pt0_), pt1(pt1_)
  { }

  void translate( double x_offset, double y_offset )
  {
    pt0.translate( x_offset, y_offset );
    pt1.translate( x_offset, y_offset );
  }

  void scale( double scale )
  {
    pt0.scale( scale );
    pt1.scale( scale );
  }

  void rotate( double angle )
  {
    pt0.rotate( angle );
    pt1.rotate( angle );
  }

};
```

```
struct Triangle
{
  Point pt0;
  Point pt1;
  Point pt2;

  Triangle()
  { }

  Triangle( Point pt0_, Point pt1_, Point pt2_ )
    : pt0(pt0_), pt1(pt1_), pt2(pt2_)
  { }

  void translate( double x_offset, double y_offset )
  {
    pt0.translate( x_offset, y_offset );
    pt1.translate( x_offset, y_offset );
    pt2.translate( x_offset, y_offset );
  }

  void scale( double scale )
  {
    pt0.scale( scale );
    pt1.scale( scale );
    pt2.scale( scale );
  }

  void rotate( double angle )
  {
    pt0.rotate( angle );
    pt1.rotate( angle );
    pt2.rotate( angle );
  }

};
```

2. Groups, Canvases, Drawings

- We wish to declare a new `Group` class which is a composition of points, lines, and triangles
  - Member fields to store shapes
  - Member functions to add and transform shapes

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

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

2.1. 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 )
{
  Group group;
  group.add( Point(1,2) );
  group.m_points[0].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 Group
{
 public:

  Group()
   : m_points_size(0), m_lines_size(0), m_triangles_size(0)
  { }

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

  ...

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

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

- We are free to change how we store shapes
- We could use dynamically resizable lists or vectors

2.2. C++ Incomplete Types

- We want a `Group` to also compose other `Groups`
- So we might try something like the following ...

```
1  class Group
2  {
3   public:
4
5    void add( const Group& group )
6    {
7      assert( m_groups_size < 16 );
8      m_groups[m_groups_size] = group; m_groups_size++;
9    }
10
11   void translate( double x_offset, double y_offset )
12   {
13     ...
14     for ( int i = 0; i < m_groups_size; i++ )
15       m_groups[i].translate( x_offset, y_offset );
16   }
17   ...
18
19  private:
20   ...
21   Group m_groups[16];
22   int   m_groups_size;
23 };
```

- This code will not compile since a `Group` contains `Groups` the compiler cannot figure out how to layout the `Group` class

- How big is a `Group`? It depends on how big `Group` is!
- On line 22, `Group` is still an incomplete type

- We can instead store a list of `Group` pointers
- Pointers are always the same size regardless of their type

```
class Group
{
 public:

  ~Group()
  {
    for ( int i = 0; i < m_group_ptrs_size; i++ )
      delete m_group_ptrs[i];
  }

  void add( const Group& group )
  {
    assert( m_group_ptrs_size < 16 );
    m_group_ptrs[m_group_ptrs_size] = new Group( group );
    m_group_ptrs_size++;
  }

  void translate( double x_offset, double y_offset )
  {
    ...
    for ( int i = 0; i < m_group_ptrs_size; i++ )
      m_group_ptrs[i]->translate( x_offset, y_offset );
  }
  ...

 private:
  ...
  Group* m_group_ptrs[16];
  int    m_group_ptrs_size;
};
```

- Default copy constructor will just copy the *pointers*
- Default copy constructor will not copy what pointers *point to*
- Rule of Three: If you define a copy constructor, destructor, or assignment operator, then you should probably define all three

```
class Group
{
  ...
  // Explicit copy constructor
  Group( const Group& group )
  {
    // Same as default copy constructor
    m_points_size    = group.m_points_size;
    for ( int i = 0; i < m_points_size; i++ )
      m_points[i] = group.m_points[i];

    // Same as default copy constructor
    m_lines_size     = group.m_lines_size;
    for ( int i = 0; i < m_lines_size; i++ )
      m_lines[i] = group.m_lines[i];

    // Same as default copy constructor
    m_triangles_size = group.m_triangles_size;
    for ( int i = 0; i < m_triangles_size; i++ )
      m_triangles[i] = group.m_triangles[i];

    // Recursively call copy constructor on all groups
    // Recreate tree of dynamically allocated groups

    m_group_ptrs_size = group.m_group_ptrs_size;
    for ( int i = 0; i < m_group_ptrs_size; i++ )
      m_group_ptrs[i] = new Group( *group.m_group_ptrs[i] );
  }

  // Explicit assignment operator: first delete, then new
  ...
};
```

```
class Canvas
{
 public:
  Canvas()
  {
    for ( size_t i = 0; i < 31; i++ ) {
      for ( size_t j = 0; j < 31; j++ ) {
        m_frame[j][i] = false;
      }
    }
  }

  void mark( int x, int y )
  {
    if ( ( x < -15 ) || ( x > 15 ) )
      return;

    if ( ( y < -15 ) || ( y > 15 ) )
      return;

    m_frame[x+15][y+15] = true;
  }

  void display() const
  {
    // use printf to display frame
  }

 private:
  bool m_frame[31][31];
};
```

- We need to add a new `draw` member function to each shape

```
class Line
{
  ...
  void Line::draw( Canvas* canvas ) const
  {
    assert( canvas != NULL );
    double dist = diagonal_distance( pt0, pt1 );
    for ( int i = 0; i <= dist; i++ ) {
      double t = ( i == 0 ) ? 0 : i / dist;
      Point pt = lerp_point( pt0, pt1, t );
      pt.draw( canvas );
    }
  }

 // Private helper functions
 private:

  static Point
  lerp_point( const Point& pt0, const Point& pt1, double t );

  static double
  diagonal_distance( const Point& pt0, const Point& pt1 );
};

class Triangle
{
  ...
  void draw( Canvas* canvas ) const
  {
    Line line0( pt0, pt1 );
    Line line1( pt1, pt2 );
    Line line2( pt2, pt0 );
    line0.draw( canvas );
    line1.draw( canvas );
    line2.draw( canvas );
  }
};
```

- `Drawing` simply composes a top-level `Group` and `Canvas`

```
1  int main( void )
2  {
3    srand( time(NULL) );
4
5    // Create a group of lines forming a star
6
7    Group star;
8    for ( int i = 0; i < 8; i++ )
9      star.add( Line( Point(0,0), Point(0,3) ) % (i*45) );
10
11   // Randomly place stars on a drawing
12
13   Drawing drawing;
14   for ( int i = 0; i < 6; i++ ) {
15     int x_offset = ( rand() % 30 ) - 15;
16     int y_offset = ( rand() % 30 ) - 15;
17     drawing.add( star + Point( x_offset, y_offset ) );
18   }
19   drawing.display();
20
21   return 0;
22 }
```

`http://cpp.sh/6tgtb`

3. Lists

- Object-oriented programming can greatly simplify implementing and using data structures and algorithms
- Recall the singly linked list data structure implemented in C

```c
typedef struct _node_t
{
  int             value;
  struct _node_t* next_p;
}
node_t;
```

```c
typedef struct
{
  node_t* head_p;
}
list_t;
```

```c
void list_construct  ( list_t* list_p );
void list_destruct   ( list_t* list_p );
void list_push_front ( list_t* list_p, int v );
```

3.1. Basic List Interface and Implementation

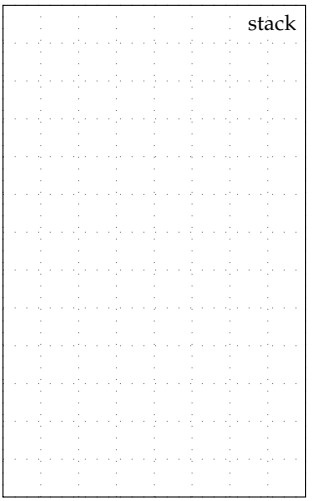
```c
class List
{
 public:

  List();                     // constructor
  ~List();                    // destructor
  void push_front( int v );   // member function

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

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

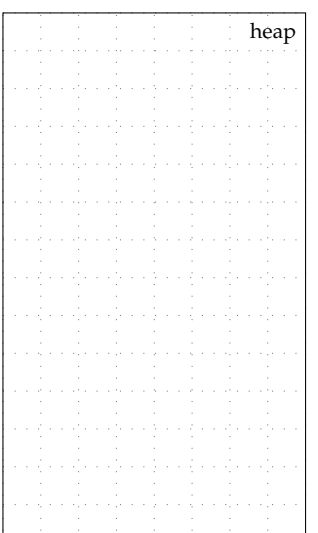
- Notice the syntax used for separating member function *declarations* from member function *definitions* implement a
- `nullptr` is a new C++11 keyword for a null pointer

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

- Rule of threes means we also need to declare and define a copy constructor and an overloaded assignment operator
- Consider refactoring out code to `clean` a list and code to `copy` a list; reuse these private functions in the destructor, copy constructor, and overloaded assignment operator

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





3.2. Iterator-Based List Interface and Implementation

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

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

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

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

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

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

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

```
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;
}
```

```
List::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++)
List::Itr operator++( List::Itr& itr, int )
{
  List::Itr itr_tmp = itr; itr.next(); return itr_tmp;
}

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

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

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

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

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

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

3.3. `auto` and Range-Based Loops

- C++11 `auto` keyword will automatically infer type from initializer

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

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

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

- C++11 range-based loops work for regular arrays too

```
int a[] = { 1, 2, 3, 4, 5 };
for ( int v : a ) {
  printf( "%d\n", v );
}
```

3.4. Sorting with Iterators

4. Vectors

- Recall the fixed-size vector data structure implemented in C

```
typedef struct
{
  int*   data;
  size_t maxsize;
  size_t size;
}
vector_t;

void vector_construct  ( vector_t* vec_p, size_t maxsize );
void vector_destruct   ( vector_t* vec_p );
void vector_push_front ( vector_t* vec_p, int v );
```

4.1. Basic Vector Interface and Implementation

```
class Vector
{
 public:

  Vector( size_t maxsize );
  ~Vector();
  void push_front( int v );

  int*   m_data;
  size_t m_maxsize;
  size_t m_size;
};
```

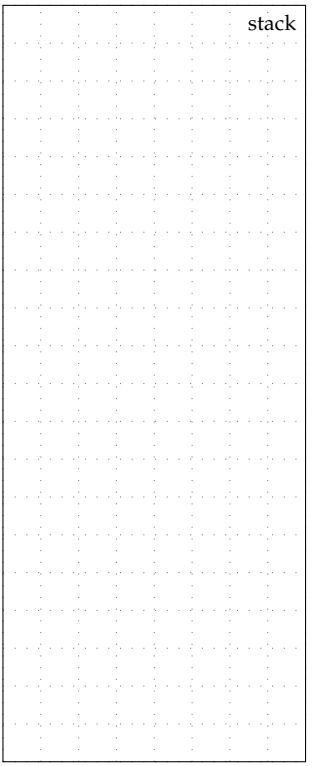
```
Vector::Vector( size_t maxsize )
{
  m_data    = new int[maxsize];
  m_maxsize = maxsize;
  m_size    = 0;
}

Vector::~Vector()
{
  delete[] m_data;
  m_data = nullptr;
}

void Vector::push_front( int v )
{
  assert( (m_maxsize - m_size) >= 1 );
  int prev_value = v;
  for ( size_t i = 0; i <= m_size; i++ )
    std::swap( prev_value, m_data[i] );
  m_size += 1;
}
```

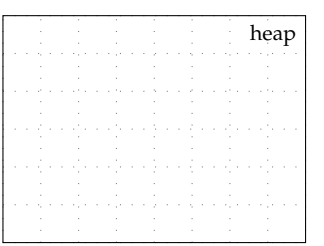
- Cannot use range-based loop, size is not compile time constant

```
  // will not compile
  for ( int&  v : m_data )
    std::swap( prev_value, v );
```

- Classes included an implicitly defined copy constructor, destructor, and assignment operator if they are not specified
- Implicitly definitions perform a shallow copy
- Rule of Three: If you define one (e.g., to do a deep copy) you should probably define all three appropriately

```
1  int main( void )
2  {
3    Vector vec(4);
4    vec.push_front(12);
5    vec.push_front(11);
6    vec.push_front(10);
7
8    for ( size_t i = 0;
9          i <= vec.m_size; i++ ) {
10     printf( "%d\n",
11             vec.m_data[i] );
12   }
13
14   return 0;
15 }
```





4.2. Iterator-Based Vector Interface and Implementation

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

```
class Vector
{
 public:

  class Itr
  {
   public:
    Itr( int* m_data, size_t idx );
    void next();
    int& get();
    bool eq( const Itr& itr ) const;

   private:
    friend class Vector;
    int*   m_data;
    size_t m_idx;
  };

  Itr begin();
  Itr end();

  ...

 private:
  int*   m_data;
  size_t m_maxsize;
  size_t m_size;
};
```

```
Vector::Itr::Itr( int* data, size_t idx )
  : m_data(data), m_idx(idx)
{ }

void Vector::Itr::next()
{
  m_idx++;
}

int& Vector::Itr::get()
{
  return m_data[m_idx];
}

bool Vector::Itr::eq( const Itr& itr ) const
{
  return ( (m_data == itr.m_data) && (m_idx == itr.m_idx) );
}

Vector::Itr Vector::begin() { return Itr(m_data,0);      }
Vector::Itr Vector::end()   { return Itr(m_data,m_size); }
```

- Can use vector iterators just like list iterators

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

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

4.3. Sorting with Iterators

```
1  void Vector::sort()
2  {
3    for ( Itr itr0 = begin(); itr0 != end(); itr0++ ) {
4      int max = *itr0;
5      for ( Itr itr1 = begin(); itr1 != itr0; itr1++ ) {
6        if ( max < *itr1 )
7          std::swap( temp, *itr );
8      }
9      *itr0 = max;
10   }
11 }
```

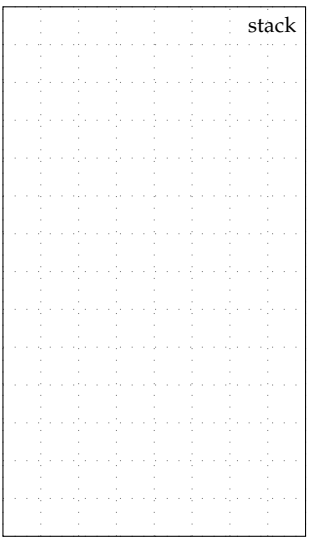
- Code is identical to sort for lists
- Can we refactor this code into a single sorting implementation?

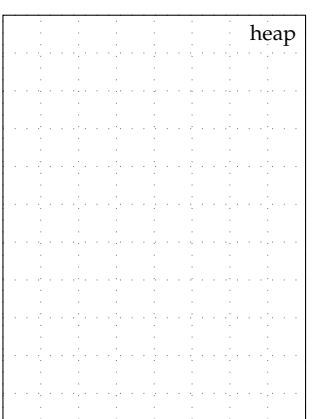
5. Revisiting Strings and Standard Output

- C strings are tedious and error prone
- Standard C++ library includes object-oriented string class

```
1  #include <string>
2
3  std::string
4  sw2hw( const std::string& str )
5  {
6    std::string tmp = str;
7    size_t pos = tmp.find("2400");
8    tmp.replace( pos, 4, "2300" );
9    return tmp;
10 }
11
12 int main( void )
13 {
14   std::string str0("ECE");
15   std::string str1 = str0;
16   bool eq = ( str0 == str1 );
17   printf("%d\n",eq);
18
19   std::string str2 = "2400";
20   std::string str3 =
21     str1 + " " + str2;
22   printf("%s\n",str2.c_str());
23
24   std::string str4 = sw2hw( str3 );
25   printf("%s\n",str4.c_str());
26
27   return 0;
28 }
```

http://cpp.sh/7yuof





- `printf` cannot automatically handle user-defined types

```
1  struct Complex
2  {
3    double real;
4    double imag;
5
6    Complex( double real_, double imag_ )
7      : real(real_), imag(imag_)
8    { }
9
10 };
11
12 int main( void )
13 {
14   Complex a(1,2);
15   printf( "%d: %f+%fi\n", 42, a.real, a.imag );
16   return 0;
17 }
```

- How can we take a more object oriented approach to I/O?

- We can use the concept of a stream which contains state about the input or output (e.g., file info, formatting info)
- Unified struct with member functions overloaded for each type?

```
1  struct ostream
2  {
3    void write( int i )
4    {
5      printf("%d",i);
6    }
7
8    void write( const char* str )
9    {
10     printf("%s",str);
11   }
12
13   void write( const Complex& complex )
14   {
15     printf("%f+%fi",complex.real,complex.imag);
16   }
17
18   // State about stream goes here
19 };
20
21 ostream cout;
22
23 int main( void )
24 {
25   cout.write( 42           );
26   cout.write( ":"          );
27   cout.write( Complex(1,2) );
28   cout.write( "\n"         );
29   return 0;
30 }
```

- Free functions overloaded for each type?

```
1  struct ostream
2  {
3    // State about stream goes here
4  };
5
6  ostream cout;
7
8  void write( ostream& os, int i )
9  {
10   printf("%d",i);
11 }
12
13 void write( ostream& os, const char* str )
14 {
15   printf("%s",str);
16 }
17
18 void write( ostream& os, const Complex& complex )
19 {
20   printf("%f+%fi",complex.real,complex.imag);
21 }
22
23 int main( void )
24 {
25   write( cout, 42           );
26   write( cout, ":"          );
27   write( cout, Complex(1,2) );
28   write( cout, "\n"         );
29   return 0;
30 }
```

- Carefully chosen operator overloaded for each type?

```
1  struct ostream
2  {
3    // State about stream goes here
4  };
5
6  ostream cout;
7
8  ostream& operator<<( ostream& os, int i )
9  {
10   printf("%d",i);
11 }
12
13 ostream& operator<<( ostream& os, const char* str )
14 {
15   printf("%s",str);
16 }
17
18 ostream& operator<<( ostream& os, const Complex& complex )
19 {
20   printf("%f+%fi",complex.real,complex.imag);
21 }
22
23 int main( void )
24 {
25   cout << 42 << ":" << Complex(1,2) << "\n";
26   return 0;
27 }
```

- I/O stream manipulators

```
struct EndOfLine
{ };

EndOfLine endl;

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

int main( void )
{
  cout << 42 << ":" << Complex(1,2) << endl;
  return 0;
}
```

- Standard C++ library provides a set of very sophisticated stream-based I/O classes
- These classes do *not* use `printf`, but the above captures the high-level idea

```
#include <iostream>

std::ostream& operator<<( std::ostream& os,
                          const Complex& complex )
{
  os << complex.real << "+" << complex.imag << "i";
}

int main( void )
{
  std::cout << 42 << ":" << Complex(1,2) << std::endl;
  return 0;
}
```

- Streams to read from standard input

```
int main( void )
{
  std::cout << "Enter a number: ";
  int num;
  std::cin >> num;
  return 0;
}
```

- Streams to write files

```
int main( void )
{
  std::ofstream fout;
  fout.open("example.txt");
  fout << 42 << ":" << Complex(1,2) << std::endl;
  fout.close();
  return 0;
}
```

- Streams to write strings

```
int main( void )
{
  std::stringstream ss;
  ss << 42 << ":" << Complex(1,2);
  std::string str = ss.str();
  std::cout << str << std::endl;
}
```