ECE 2400 Computer Systems Programming Spring 2026

Topic 12: Object-Oriented Programming

School of Electrical and Computer Engineering
Cornell University

revision: 2026-03-22-20-14

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

1. C++ Classes

1.1. Approximating C++ Classes in C

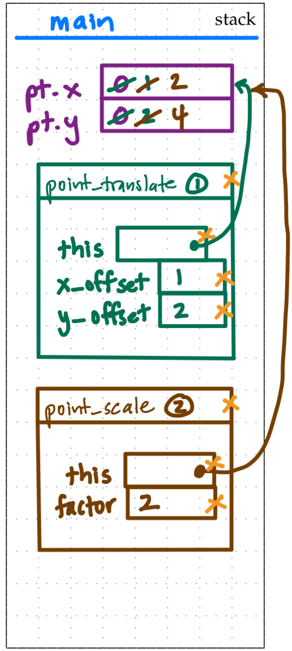
- Perfectly possible to use object-oriented programming in C
- Possible, but not pretty

```
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;
}
```

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

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

* point_translate code is on previous page

* point_scale code is on previous page



1.2. C++ Member Functions

- C++ `struct` has both member fields and member functions
  - Functions are defined *within* the `struct` namespace
  - Member functions are accessed using the dot (.) operator, just like fields
- Member functions have an implicit `this` pointer
  - No need to explicitly use `this` inside the function body
  - Member fields are automatically in scope within every member function
  - Member functions that do not modify any fields are marked `const`
- Being inside the `struct` namespace eliminates naming boilerplate
  - No need for the `point_` prefix on function names
  - No need for the `pt` parameter to pass the struct explicitly

```
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;
  }
};
```

Draw a state diagram corresponding to the execution of this program

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

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

stack

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

1.3. 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

Draw a state diagram corresponding to the execution of this program

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

  // default constructor
  Point()
  {
    x = 0;
    y = 0;
  }

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

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

};

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

stack

- 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 and const members

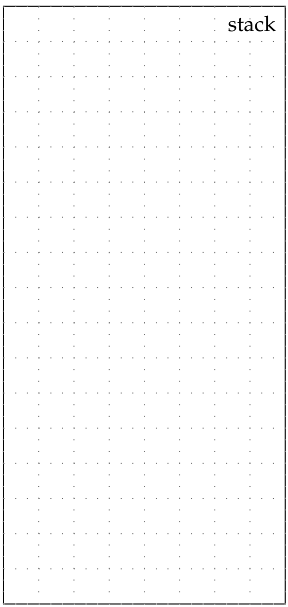
```
struct Point
{
  double x;
  double y;

  // default constructor
  Point()
  { x = 0; y = 0; }

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

  ...
};
```

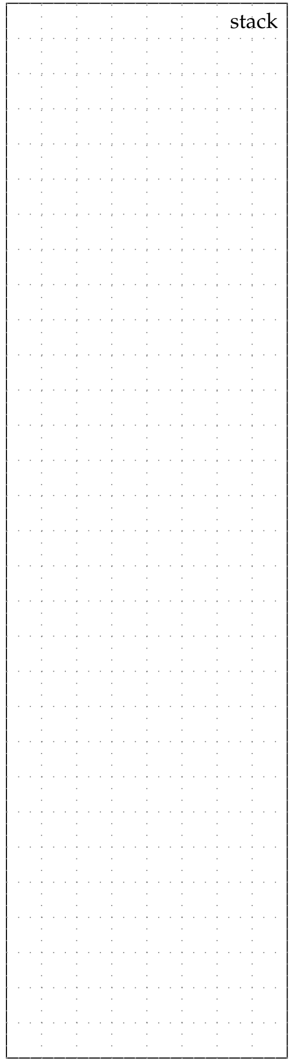
```
struct Point
{
  double x;
  double y;

  // default constructor
  Point()
   : x(0), y(0) { }

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

  ...
};
```

1.4. 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+( Point pt0,
                 Point pt1 )
{
  pt0.translate( pt1.x, pt1.y );
  return pt0;
}

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





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

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;
}
```

- 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;
```

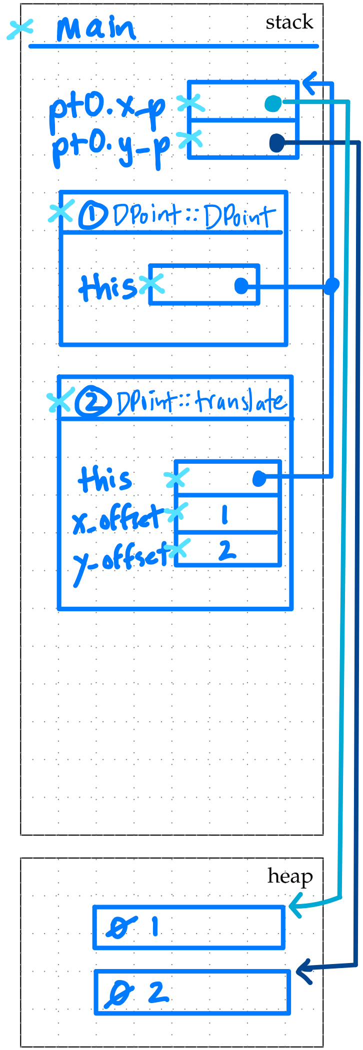
```
Point pt0(1,2);
Point pt1 = 1.5 * ( ( pt0 + Point(5,3) ) % 45 );
```

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;
    *y_p = 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, 2 );
  return 0;
}
```

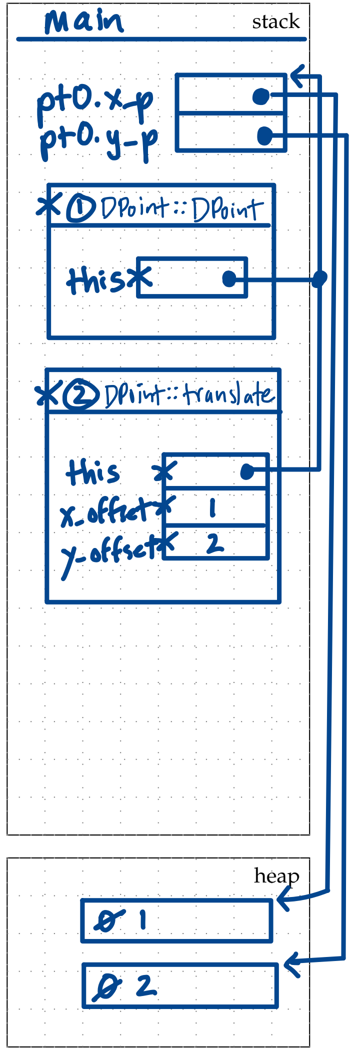
Is there a problem?



C++ Destructors

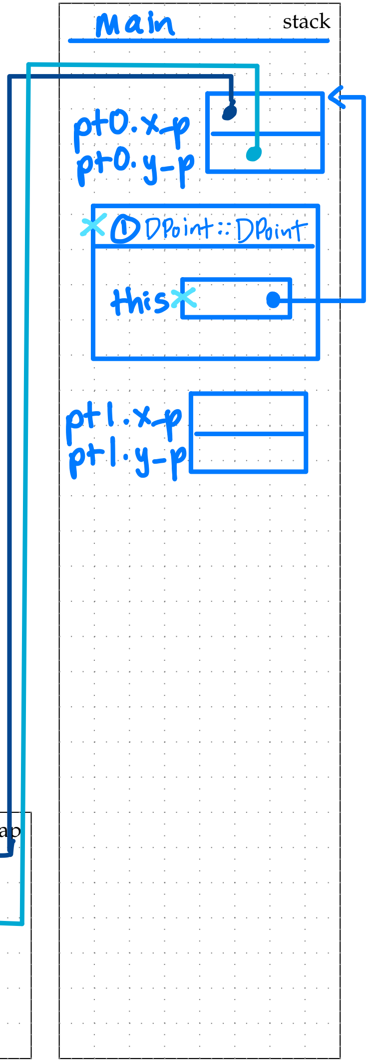
- Special member function to destroy an object

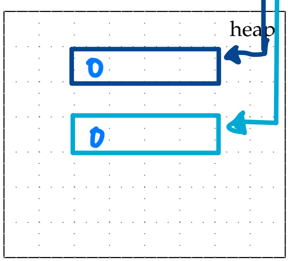
```
struct DPoint
{
  double* x_p;
  double* y_p;

  DPoint() {
    x_p  = new double;
    y_p  = new double;
    *x_p = 0;
    *y_p = 0;
  }

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

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

  ...
};

int main( void )
{
  DPoint pt0;
  pt0.translate( 1, 2 );
  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;
    *y_p = 0;
  }

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

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

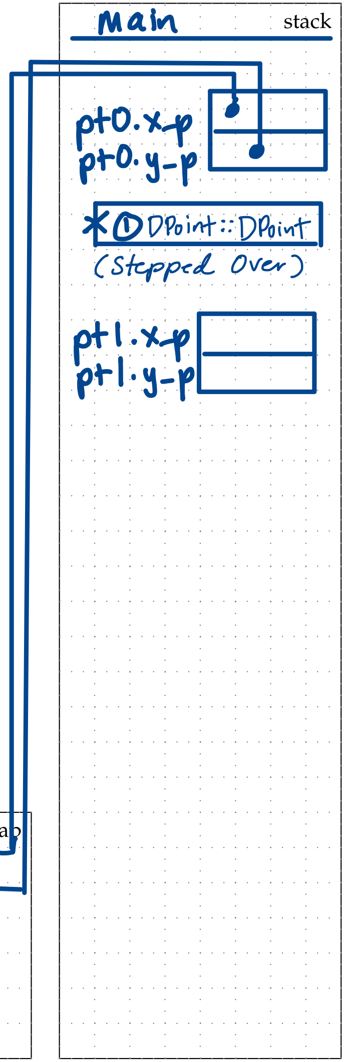


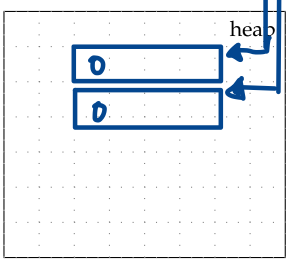


C++ Copy Constructors

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

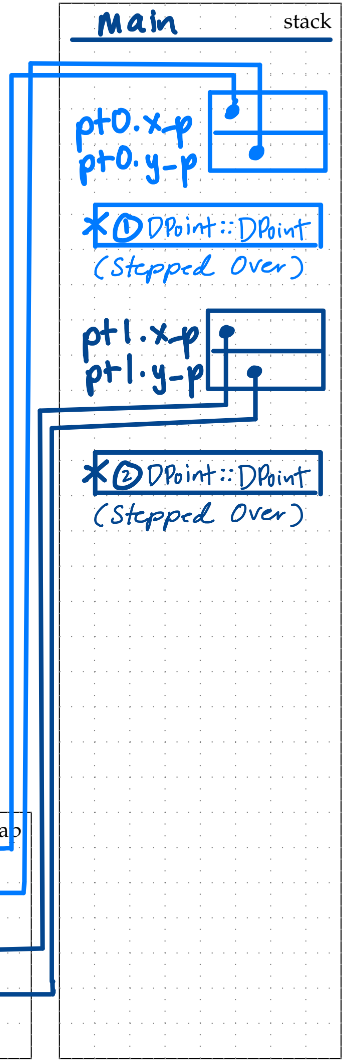
```
struct DPoint
{
  double* x_p;
  double* y_p;

  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, 2 );
  return 0;
}
```

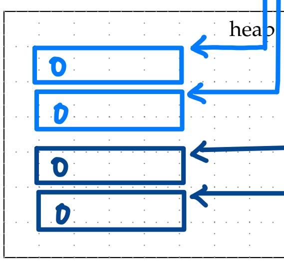




- What if we assign to an object with dynamically allocated memory?

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

  DPoint() {
    x_p  = new double;
    y_p  = new double;
    *x_p = 0;
    *y_p = 0;
  }

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

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

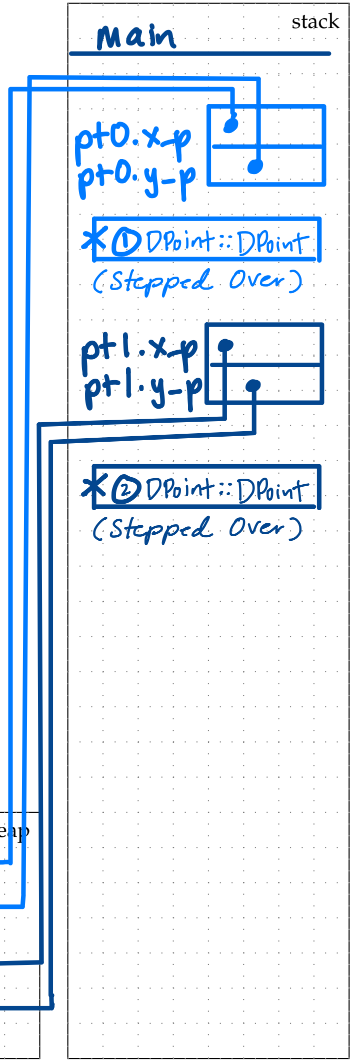


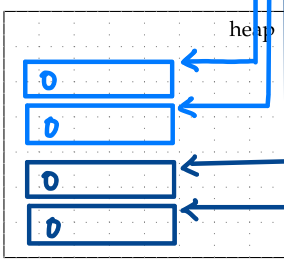


C++ Assignment Operators

- 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, 2 );
  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;
    *y_p = 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;
  }

  ...
};
```

Label all calls to the rule of three member functions

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

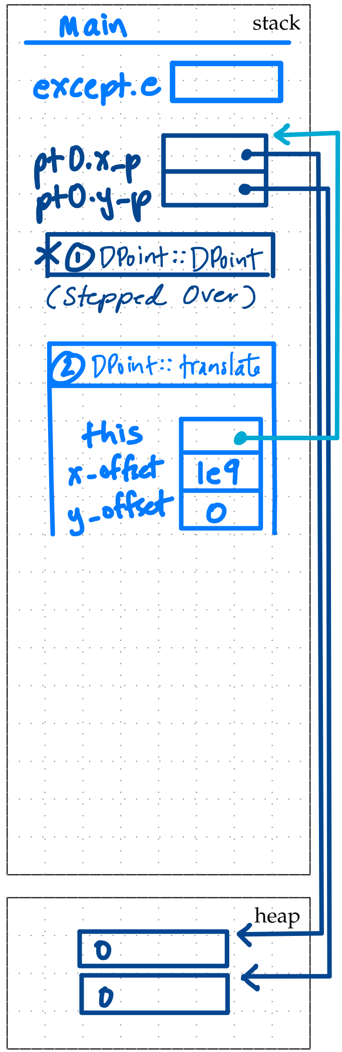
```
void ex0()
{
  Point pt0(1,2);
  Point pt1(3,4);
  pt0 = p2;
}

void ex1()
{
  DPoint pt0(1,2);
  DPoint pt1( pt0 );
  DPoint pt2 = pt1;
  pt0 = p2;
  pt0 = p2 = p1;
  pt0 = pt0;
}

DPoint foo( DPoint pt )
{
  return pt;
}

void ex2()
{
  DPoint pt0(1,2);
  DPoint pt1 = foo( pt0 );
  DPoint pt2;
  pt2 = foo( pt1 );
}

void ex3()
{
  DPoint pt0(1,2);
  DPoint pt1(3,4);
  DPoint* a = &pt0;
  DPoint* b = a;
}
```

```
void ex4()
{
  DPoint* pt0 = new DPoint(1,2);
}

void ex5()
{
  DPoint* pt0 = new DPoint(1,2);
  delete pt0;
}

void ex6()
{
  DPoint* pt0 = new DPoint[4];
  delete[] pt0;
}

struct TwoPoints
{
  DPoint pt0;
  DPoint pt1;
}

void ex7()
{
  TwoPoints pts0;
  TwoPoints pts1( pts0 );
  TwoPoints pts2 = pts1;
  pts0 = pts2;
}

void ex8()
{
  DPoint pt0;
  throw -1;
  DPoint pt1;
}
```

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;
}
```



1.6. C++ Scope-Bound Resource Management

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

```
DPoint transform( DPoint pt0, DPoint pt1, double scale )
{
  if ( scale < 0 )
    throw -1;

  DPoint pt;

  if ( scale == 0 ) {
    pt = DPoint(0,0);
  }
  else {
    DPoint pt2 = pt0 + pt1;
    DPoint pt3 = pt2 * scale;
    pt = pt3;
  }

  return pt;
}

int main( void )
{
  DPoint pt0(1,2);
  DPoint pt1(3,4);
  DPoint pt3 =
   transform( pt0, pt1, 2.0 );
  return 0;
}
```

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

1.7. 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 m_x; double m_y;

 public:
  // default constructor
  Point() { m_x = 0; m_y = 0; }

  // non-default constructor
  Point( double x, double y ) { m_x = x; m_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

2. Object-Oriented Data Structures

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

2.1. Singly Linked List Interface

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

```
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 );
...
```

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

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

  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

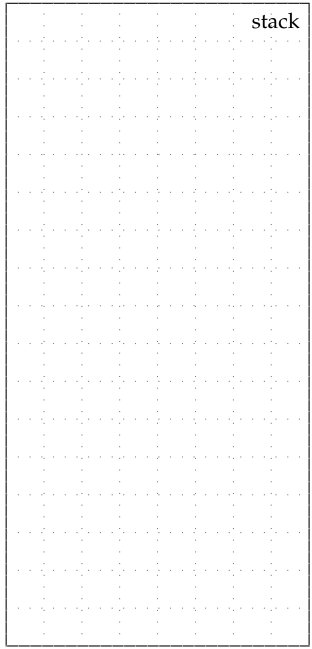
```
void slist_int_construct(
  slist_int_t* this )
{
  this->head_p = NULL;
}

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;
  }
}
```

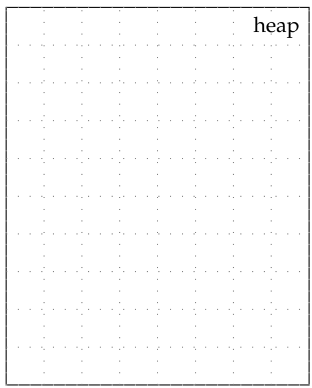
- Implementation for a C++ singly linked list data structure

```
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* curr_p
    = lst.m_head_p;
  while ( curr_p != nullptr ) {
    int value = curr_p->value;
    curr_p = curr_p->next_p;
  }

  return 0;
}
```





2.3. Iterator-Based List Interface and Implementation

- iterators improve data encapsulation and enable user to cleanly iterate through a sequence

Simple Iterator Interface: `SListInt.h`

```
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;  // `current` node, according to iterator
  };

  Itr begin();
  Itr end();

};
```

Simple Iterator Implementation: SListInt.cc

```
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);
}
```

Advanced Iterator Implementation: `SListInt.cc`

- 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 );
}
```

V0: no iterators (do everything manually)

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

V1: simple iterator

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

V2: advanced iterator

while loop:

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

for loop:

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

V3: iterator with auto

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

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

V4: C++11 range-based loops are syntactic sugar for above

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