

# 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

<b>1</b>	<b>C++ Classes</b>	<b>4</b>
1.1.	Approximating C++ Classes in C . . . . .	4
1.2.	C++ Member Functions . . . . .	6
1.3.	C++ Constructors . . . . .	8
1.4.	C++ Operator Overloading . . . . .	10
1.5.	C++ Rule of Three . . . . .	13
1.6.	C++ Scope-Bound Resource Management . . . . .	22
1.7.	C++ Data Encapsulation . . . . .	23
<b>2</b>	<b>Object-Oriented Data Structures</b>	<b>25</b>
2.1.	Singly Linked List Interface . . . . .	25
2.2.	Singly Linked List Implementation . . . . .	26
2.3.	Iterator-Based List Interface and Implementation . . . . .	29
<b>3</b>	<b>C++ Inheritance</b>	<b>33</b>
3.1.	C++ Implementation Inheritance . . . . .	35
3.2.	From Implementation to Interface Inheritance . . . . .	39

---

3.3. C++ Interface Inheritance . . . . .	47
3.4. Revisiting Composition vs. Generalization . . . . .	50
<b>4 OO Data Structures w/Dynamic Polymorphism</b>	<b>52</b>
4.1. Singly Linked List Interface . . . . .	56
4.2. Singly Linked List Implementation . . . . .	57
<b>5 Drawing Framework Case Study</b>	<b>59</b>

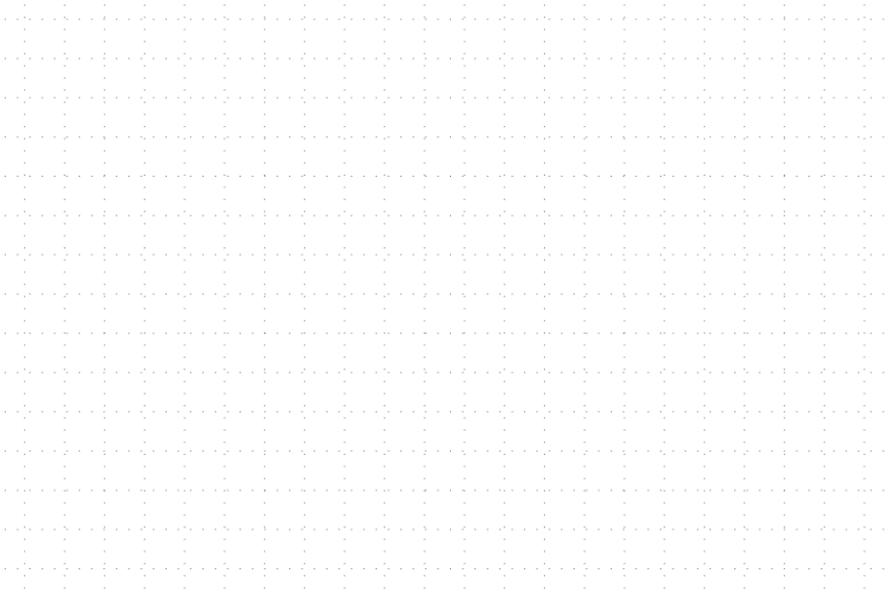
Copyright © 2026 Anne Bracy. All rights reserved. This handout was prepared by Prof. Anne Bracy at Cornell University for ECE 2400 / ENGRD 2140 Computer Systems Programming (derived from previous handouts prepared and copyrighted by Prof. Christopher Batten). Download and use of this handout is permitted for individual educational non-commercial purposes only. Redistribution either in part or in whole via both commercial or non-commercial means requires written permission.

---

## Object-oriented programming

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

### Example class diagram for animals



## 1. C++ Classes

### 1.1. Approximating C++ Classes in C

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

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

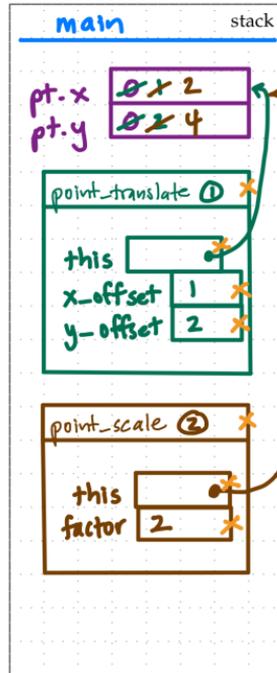
```

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

```

\*point\_translate code is on previous page

\*point\_scale code is on previous page



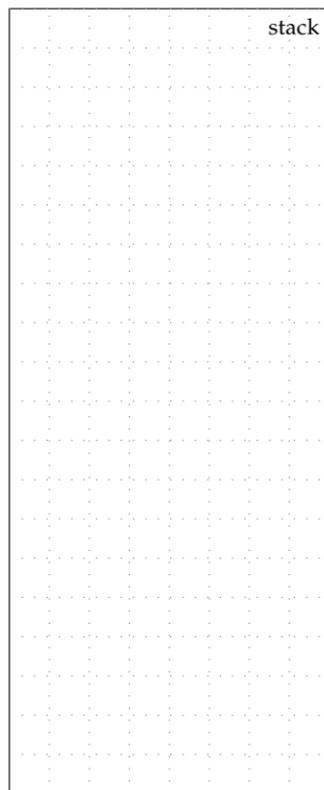
## 1.2. C++ Member Functions

- 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

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

Draw a state diagram corresponding to the execution of this program

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



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

### 1.3. C++ Constructors

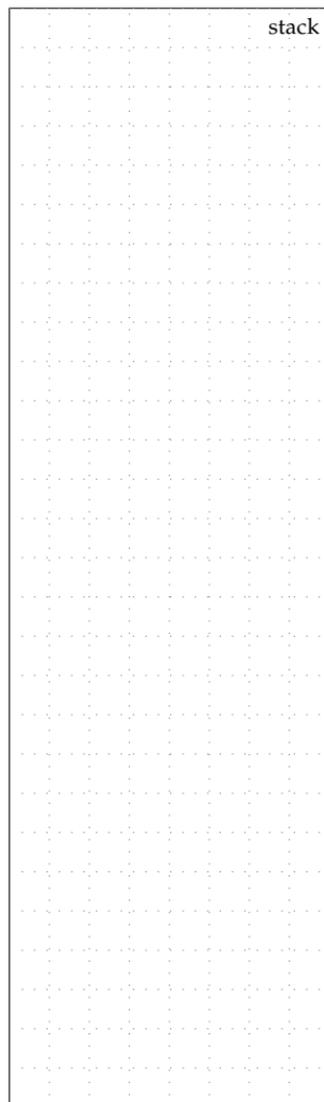
- We want to avoid the user from directly accessing member fields
- We want to ensure an object is always initialized to a known state
- In C, we used `foo_construct`
- In C++, we could add a `construct` member function

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

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

Draw a state diagram corresponding to the execution of this program

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



- Constructors automatically called with `new`

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

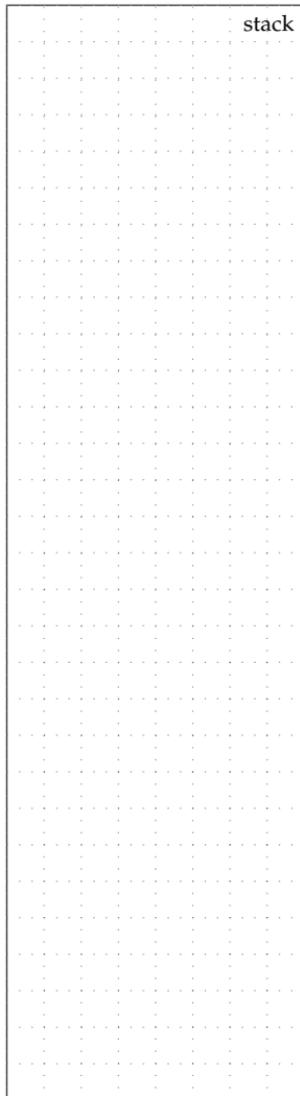
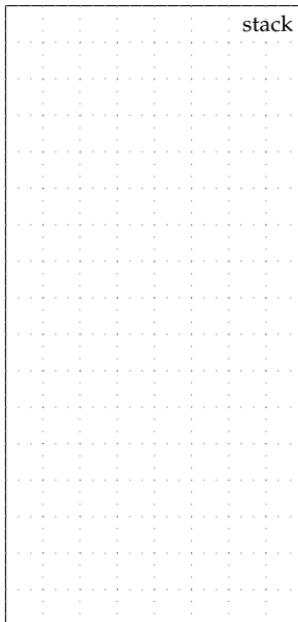
- Initialization lists initialize members before body of constructor
  - Avoids creating a temporary default object
  - Required for initializing reference and const members

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

## 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)

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



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

- Operator overloading enables elegant syntax for user-defined types

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

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

## 1.4. C++ Rule of Three

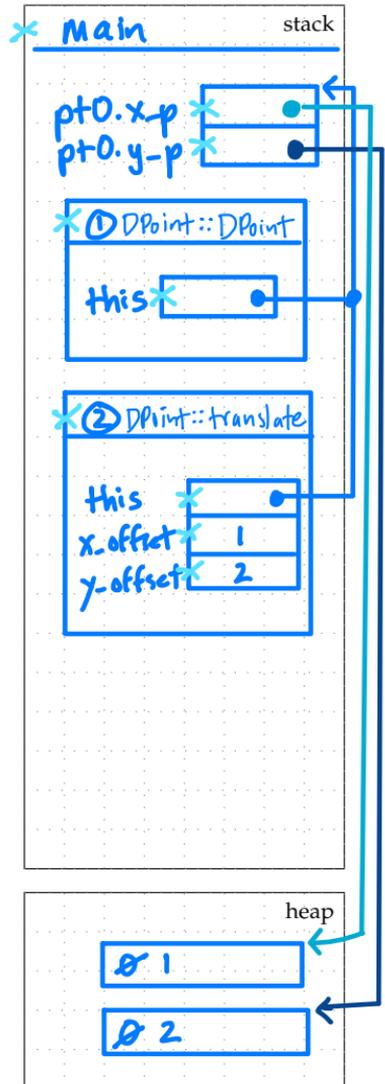
- What if point coordinates are allocated on the heap?

```

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

```

Is there a problem?



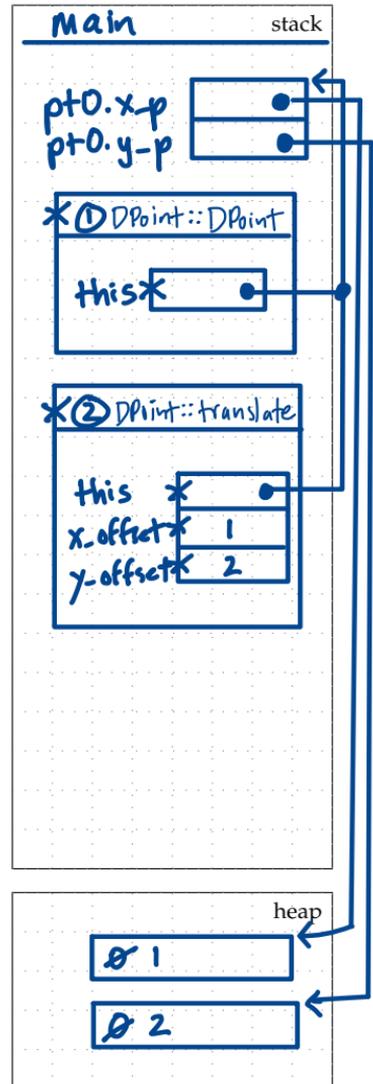
## C++ Destructors

- Special member function to destroy an object

```

01 struct DPoint
02 {
03     double* x_p;
04     double* y_p;
05
06     DPoint() {
07         x_p = new double;
08         y_p = new double;
09         *x_p = 0;
10         *y_p = 0;
11     }
12
13     ~DPoint() {
14         delete x_p;
15         delete y_p;
16     }
17
18     void translate( double x_offset,
19                   double y_offset )
20     {
21         *x_p += x_offset;
22         *y_p += y_offset;
23     }
24
25     ...
26 };
27
28 int main( void )
29 {
30     DPoint pt0; ①
31     pt0.translate( 1, 2 ); ②
32     return 0;
33 }

```

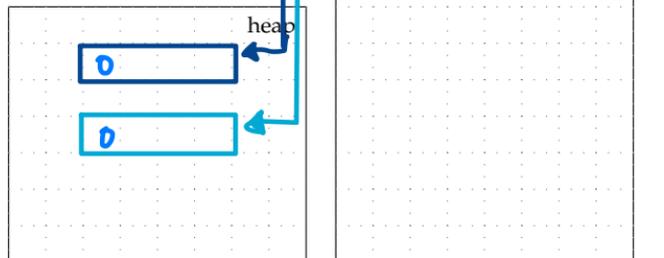


- What if we copy an object with dynamically allocated memory?

```

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

```



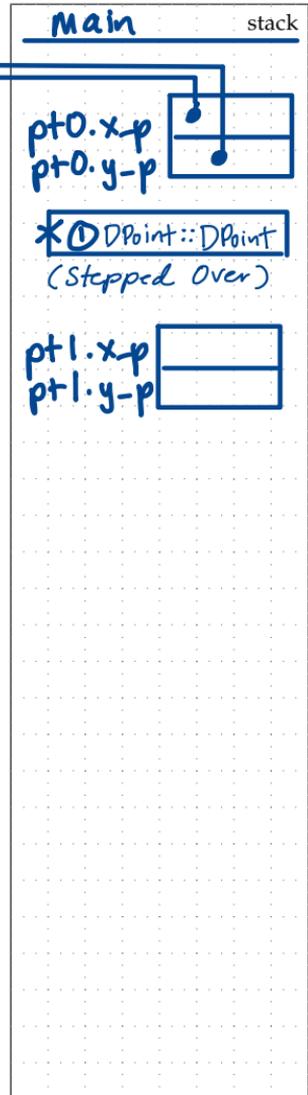
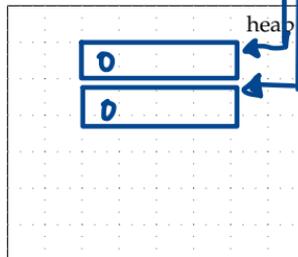
## C++ Copy Constructors

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

```

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

```

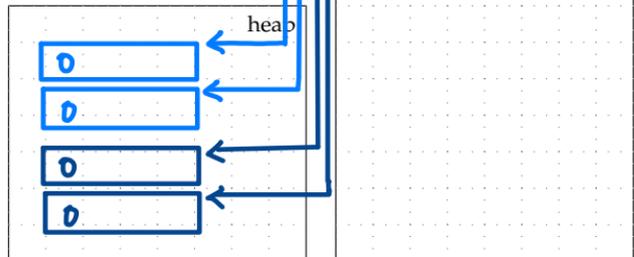


- What if we assign to an object with dynamically allocated memory?

```

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

```



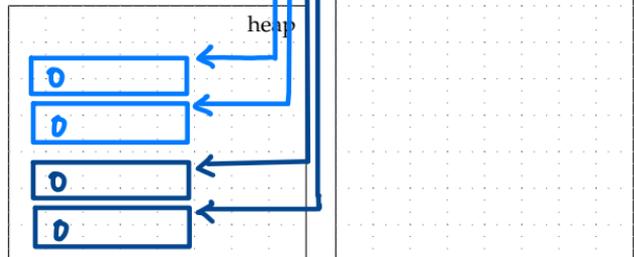
## C++ Assignment Operators

- An overloaded assignment operator will be called for assignment

```

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

```



## C++ Rule of Three

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

```
1  struct DPoint
2  {
3      double* x_p;
4      double* y_p;
5
6      DPoint()
7      {
8          x_p = new double;
9          y_p = new double;
10         *x_p = 0;
11         *y_p = 0;
12     }
13
14     DPoint( double x, double y )
15     {
16         x_p = new double;
17         y_p = new double;
18         *x_p = x;
19         *y_p = y;
20     }
21
22     DPoint( const DPoint& pt )
23     {
24         x_p = new double;
25         y_p = new double;
26         *x_p = *pt.x_p;
27         *y_p = *pt.y_p;
28     }
29
30     ~DPoint()
31     {
32         delete x_p;
33         delete y_p;
34     }
35
36     DPoint&
37     operator=( const DPoint& pt )
38     {
39         if ( this != &pt ) {
40             *x_p = *pt.x_p;
41             *y_p = *pt.y_p;
42         }
43         return *this;
44     }
45
46     ...
47 };
```

## Label all calls to the rule of three member functions

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

```

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

```

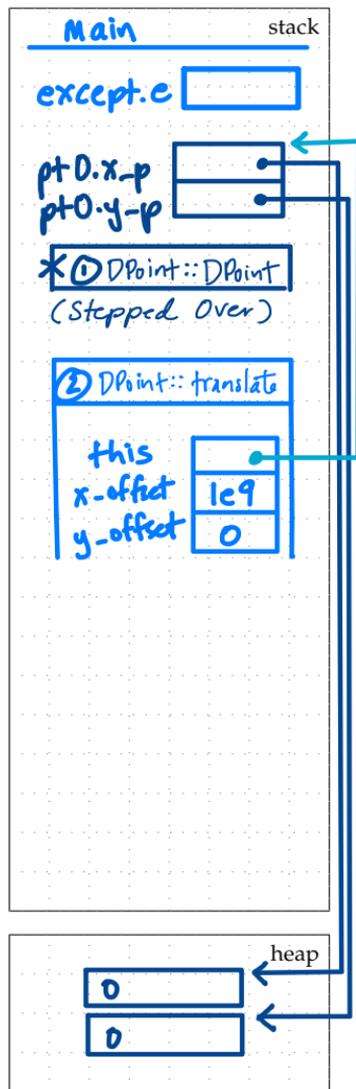
## C++ Exceptions and Destructors

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

```

01 struct DPoint
02 {
03     double* x_p;
04     double* y_p;
05
06     ~DPoint() {
07         delete x_p;
08         delete y_p;
09     }
10
11 void translate( double x_offset,
12               double y_offset )
13 {
14     if ( (x_offset > 100)
15         || (y_offset > 100) )
16         throw 42;
17     *x_p += x_offset;
18     *y_p += y_offset;
19 }
20
21 ...
22 };
23
24 int main( void )
25 {
26     try {
27         DPoint pt0; ①
28         pt0.translate( 1e9, 0 ) ②
29     }
30     catch ( int e ) {
31         return e;
32     }
33     return 0;
34 }

```



## 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

```

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

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

## 1.7. 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

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

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

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

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

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

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

## 2. Object-Oriented Data Structures

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

### 2.1. Singly Linked List Interface

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

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

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

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

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

## 2.2. Singly Linked List Implementation

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

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

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

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

- Implementation for a C singly linked list data structure

```

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

```

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

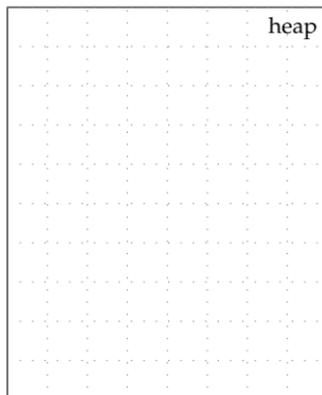
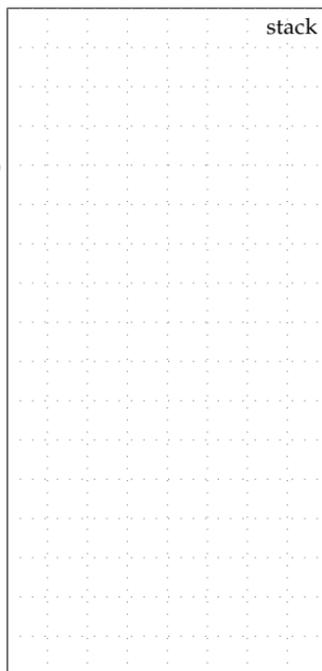
```

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

```

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

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



## 2.3. Iterator-Based List Interface and Implementation

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

### Simple Iterator Interface: SListInt.h

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

**Simple Iterator Implementation: SListInt.cc**

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

**Advanced Iterator Implementation: SListInt.cc**

- use operator overloading to improve iterator syntax

```
1 // postfix increment operator (itr++)
2 SListInt::Itr operator++( SListInt::Itr& itr, int )
3 {
4     SListInt::Itr itr_tmp = itr;
5     itr.next();
6     return itr_tmp;
7 }
8
9 // prefix increment operator (++itr)
10 SListInt::Itr& operator++( SListInt::Itr& itr )
11 {
12     itr.next();
13     return itr;
14 }
15
16 // dereference operator (*itr)
17 int& operator*( SListInt::Itr& itr )
18 {
19     return itr.get();
20 }
21
22 // not-equal operator (itr0 != itr1)
23 bool operator!=( const SListInt::Itr& itr0,
24                 const SListInt::Itr& itr1 )
25 {
26     return !itr0.eq( itr1 );
27 }
```

**V0: no iterators** (do everything manually)

```

1 SListInt::Node* curr_p = lst.m_head_p;
2 while ( curr_p != nullptr ) {
3     int value = curr_p->value;
4     printf( "%d\n", value );
5     curr_p = curr_p->next_p;
6 }

```

**V1: simple iterator**

```

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

```

**V2: advanced iterator****while loop:**

```

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

```

**for loop:**

```

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

```

**V3: iterator with auto**

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

```

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

```

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

```

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

```