ECE 2400 Computer Systems Programming Spring 2025

Topic 12: Object-Oriented Programming

School of Electrical and Computer Engineering
Cornell University

revision: 2025-03-19-12-18

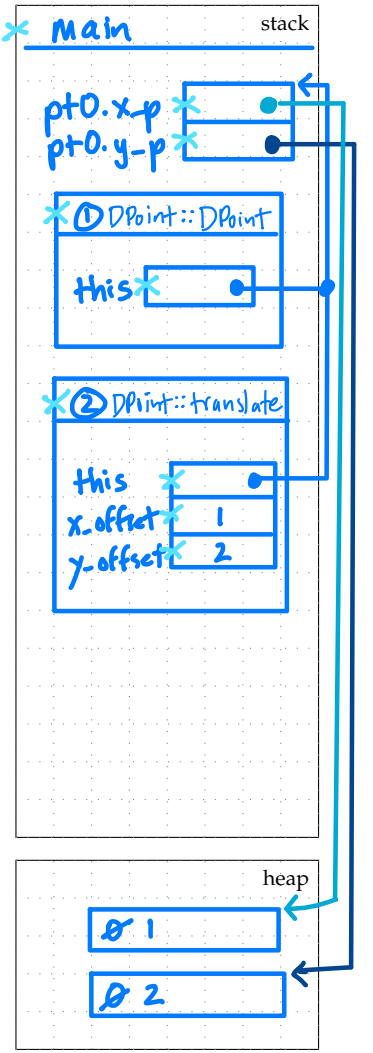
(staged for lectures)

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; (1)
25   pt0.translate( 1, 2 ); (2)
26   return 0;
27 }
```

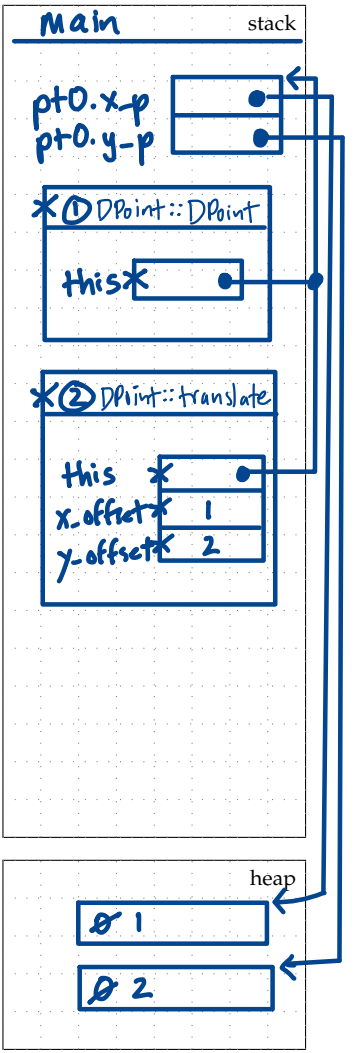
Is there a problem?



C++ Destructors

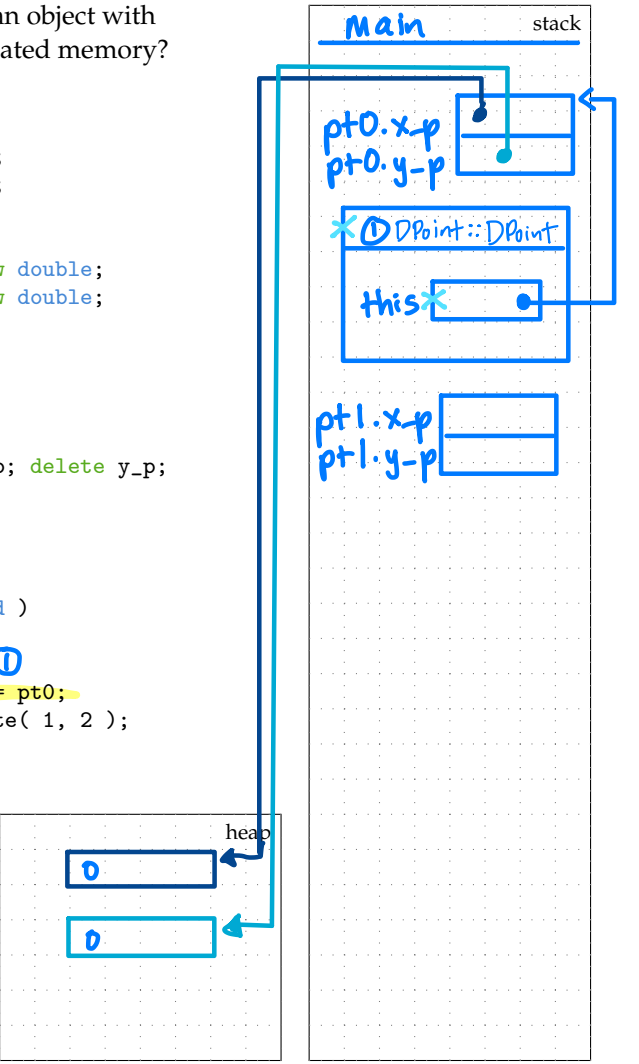
- Special member function to destroy an object

```
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?

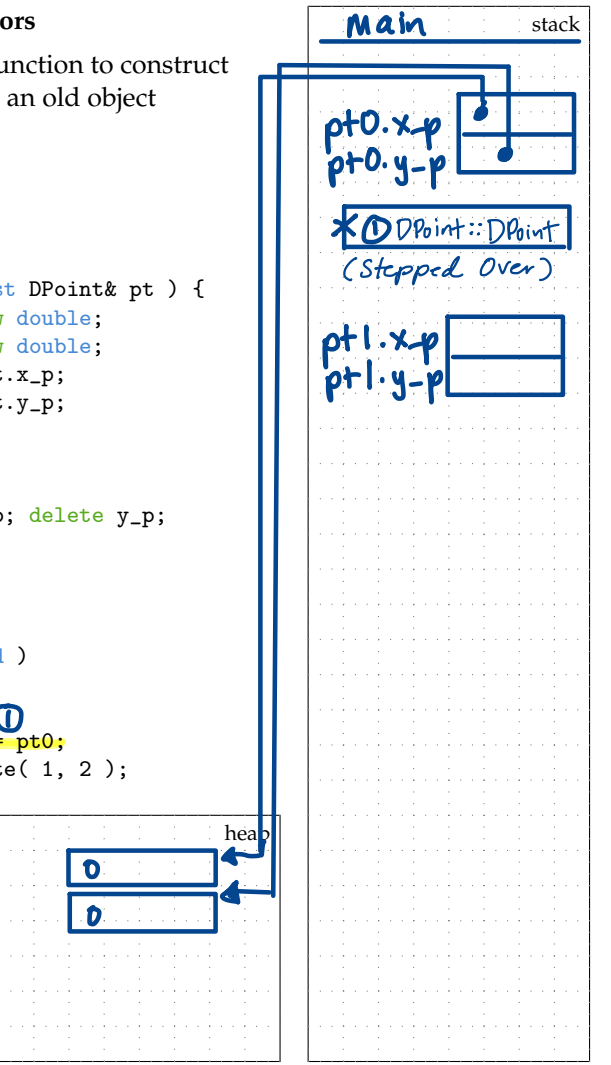
```
01 struct DPoint
02 {
03   double* x_p;
04   double* y_p;
05
06   DPoint() {
07     x_p  = new double;
08     y_p  = new double;
09     *x_p = 0;
10     *y_p = 0;
11   }
12
13   ~DPoint() {
14     delete x_p; delete y_p;
15   }
16   ...
17 };
18
19 int main( void )
20 {
21   DPoint pt0;
22   DPoint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



C++ Copy Constructors

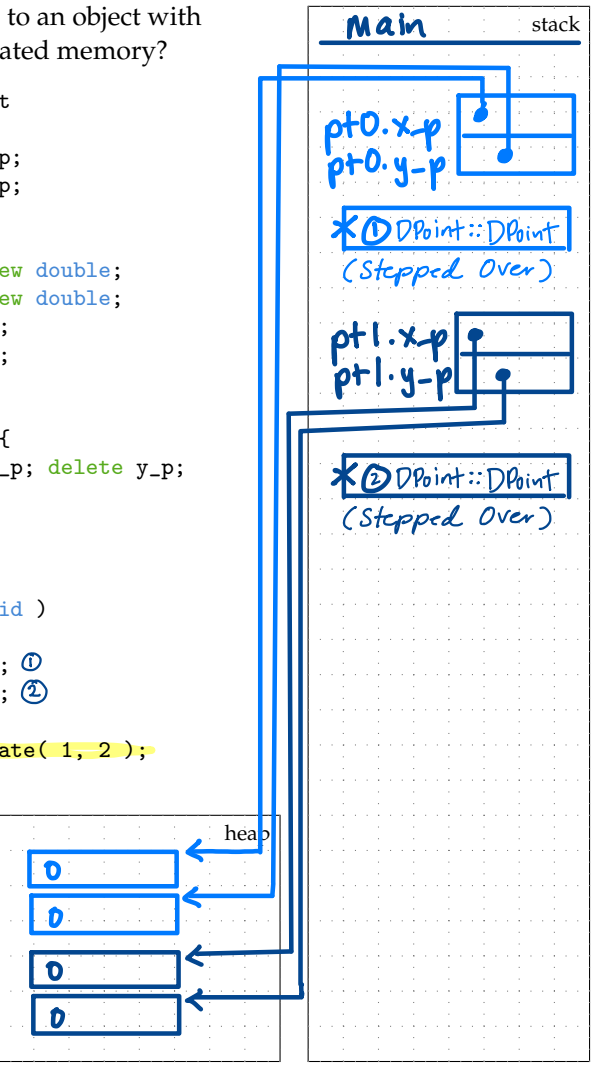
- Special member function to construct a new object from an old object

```
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; (1)
22   DPoint pt1 = pt0;
23   pt0.translate( 1, 2 );
24   return 0;
25 }
```



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

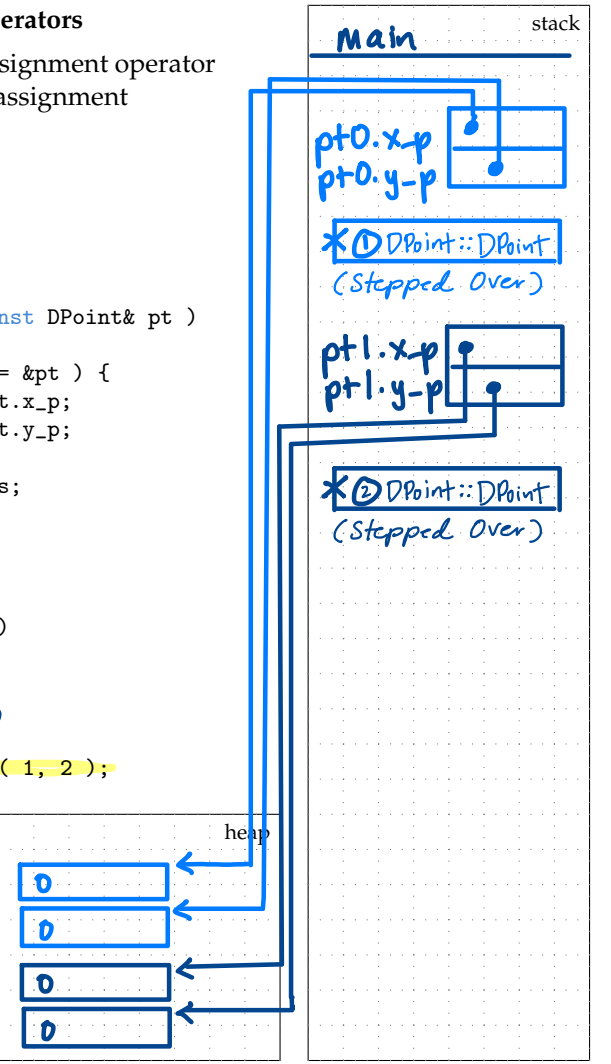
```
01 struct DPoint
02 {
03   double* x_p;
04   double* y_p;
05
06   DPoint() {
07     x_p  = new double;
08     y_p  = new double;
09     *x_p = 0;
10     *y_p = 0;
11   }
12
13   ~DPoint() {
14     delete x_p; delete y_p;
15   }
16   ...
17 };
18
19 int main( void )
20 {
21   DPoint pt0; ①
22   DPoint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

```
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

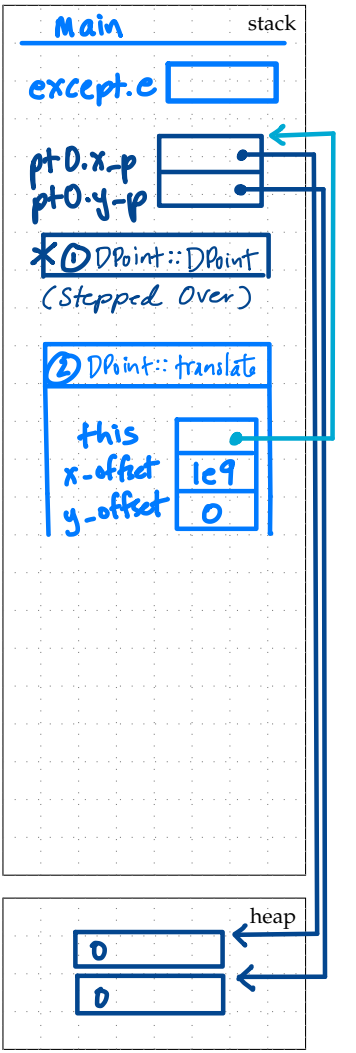
- Only label *non-trivial* destruction, initialization, assignment
- D = destructor, CC = copy constructor, AO = assignment operator

```
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 void ex4()
39 {
40   DPoint* pt0 = new DPoint(1,2);
41 }
42
43 void ex5()
44 {
45   DPoint* pt0 = new DPoint(1,2);
46   delete pt0;
47 }
48
49 void ex6()
50 {
51   DPoint* pt0 = new DPoint[4];
52   delete[] pt0;
53 }
54
55 struct TwoPoints
56 {
57   DPoint pt0;
58   DPoint pt1;
59 }
60
61 void ex7()
62 {
63   TwoPoints pts0;
64   TwoPoints pts1( pts0 );
65   TwoPoints pts2 = pts1;
66   pts0 = pts2;
67 }
68
69 void ex8()
70 {
71   DPoint pt0;
72   throw -1;
73   DPoint pt1;
74 }
```

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 }
```



1.5. 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

1.6. C++ Data Encapsulation

- Recall the importance of separating interface from implementation
- This is an example of abstraction
- In this context, also called information hiding, data encapsulation
  - Hides implementation complexity
  - Can change implementation without impacting users

- So far, we have relied on a *policy* to enforce data encapsulation
  - Users of a struct could still directly access member fields

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

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

```
1  struct Point
2  {
3   private:
4    double 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

```c
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

```cpp
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

```c
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

```c
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

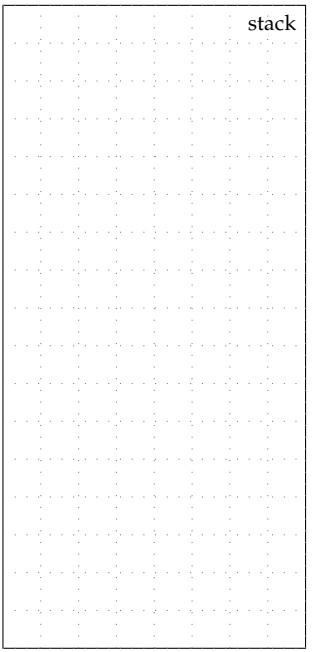
```c
void slist_int_construct(
  slist_int_t* this )
{
  this->head_p = NULL;
}

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;
  }
}
```

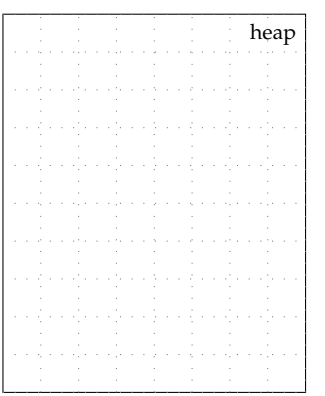
- Implementation for a C++ singly linked list data structure

```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*

```
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

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

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

```
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* curr_p
  = list.m_head_p;
while ( curr_p != nullptr ) {
  int value = curr_p->value
  printf( "%d\n", value );
  curr_p = curr_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 );
}
```