CSE 374: Programming Concepts and Tools

Spring 2026
Instructor: Megan Hazen

Today's Goals

Subject Matter

- C++ Templates
- Standard Template Library
- Smart Pointers

Your Goals

- Practice Exercises
- HW7
- Make a note sheet for the last assessment
  - Double-sided, 8.5x11

Function Templates

- We write a function that takes a datatype **T** as an argument
- Will work for any **T** for which **<** is defined.

```
#include <iostream>
#include <string>

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <typename T>   // <...> can also be  <class T>
int compare(const T& value1, const T& value2) {
  if (value1 < value2) return -1;
  if (value2 < value1) return  1;
  return 0;
}
int main(int argc, char **argv) {
  std::string h("hello"), w("world");
  std::cout << compare<int>(10, 20) << std::endl;
  std::cout << compare<std::string>(h, w) << std::endl;
  std::cout << compare<double>(50.5, 50.6) << std::endl;
  return EXIT_SUCCESS;
}
```

Function Templates – type inference

- Compiler can also infer types…. usually

```
FunctionTemplate.cc:8:14:
  warning: comparison between
  two arrays [-Warray-compare]
     8 |   if (value2 < value1)
  return  1;

FunctionTemplate.cc:8:14: note:
  use unary '+' which decays
  operands to pointers or
  '&(value2)[0] < &(value1)[0]'
  to compare the addresses
```

```
#include <iostream>
#include <string>

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <typename T>   // <...> can also be  <class T>
int compare(const T& value1, const T& value2) {
  if (value1 < value2) return -1;
  if (value2 < value1) return  1;
  return 0;
}
int main(int argc, char **argv) {
  std::string h("hello"), w("world");
  std::cout << compare(10, 20) << std::endl; // infers int
  std::cout << compare(h, w) << std::endl; // infers string
  std::cout << compare("Hello", "World") << std::endl;
        // hm…
  return EXIT_SUCCESS;}
```

Function Templates with non-types

- You can also put non-types into a template.
  - Parameter is of fixed type
  - Use comma separated list to specify template arguments.

```
// Return a pointer to an N-element heap array
// (not entirely realistic…)
template <typename T, int N>
T* valarray(const T& val) {
  T* a = new T[N];
  for (int i = 0; i < N; ++i)
    a[i] = val;
  return a;
}
int main(int argc, char** argv) {
  int* ip = valarray<int, 10>(17);
  std::string* sp = valarray<std::string, 9>("hi");
  delete[] ip;
  delete[] sp;
  return EXIT_SUCCESS; }
```

Class Templates

```
#ifndef PAIR_H_
#define PAIR_H_
#include <iostream>
#include <cstdlib>
template <typename Thing> class Pair {
 public:
  Pair() = default;  // initializes data members
  Thing get_first() const { return first_; }
  Thing get_second() const { return second_; }
  void set_first(const Thing& copyme);
  void set_second(const Thing& copyme);
  void Swap();
 private:
  Thing first_, second_;
};
#include "Pair.cc"
#endif  // PAIR_H_
```

- You can also template classes
  - Notice template before class
- This class holds a pair of any type of objects or primitives
- **`#include "Pair.cc"`** at the bottom – this is weird for a header file
  - Allows code to compile the Pair instance.
  - Could, instead, define Pair in header.
  - **`g++ -o pair usePair.cc`**

Class Templates – define code

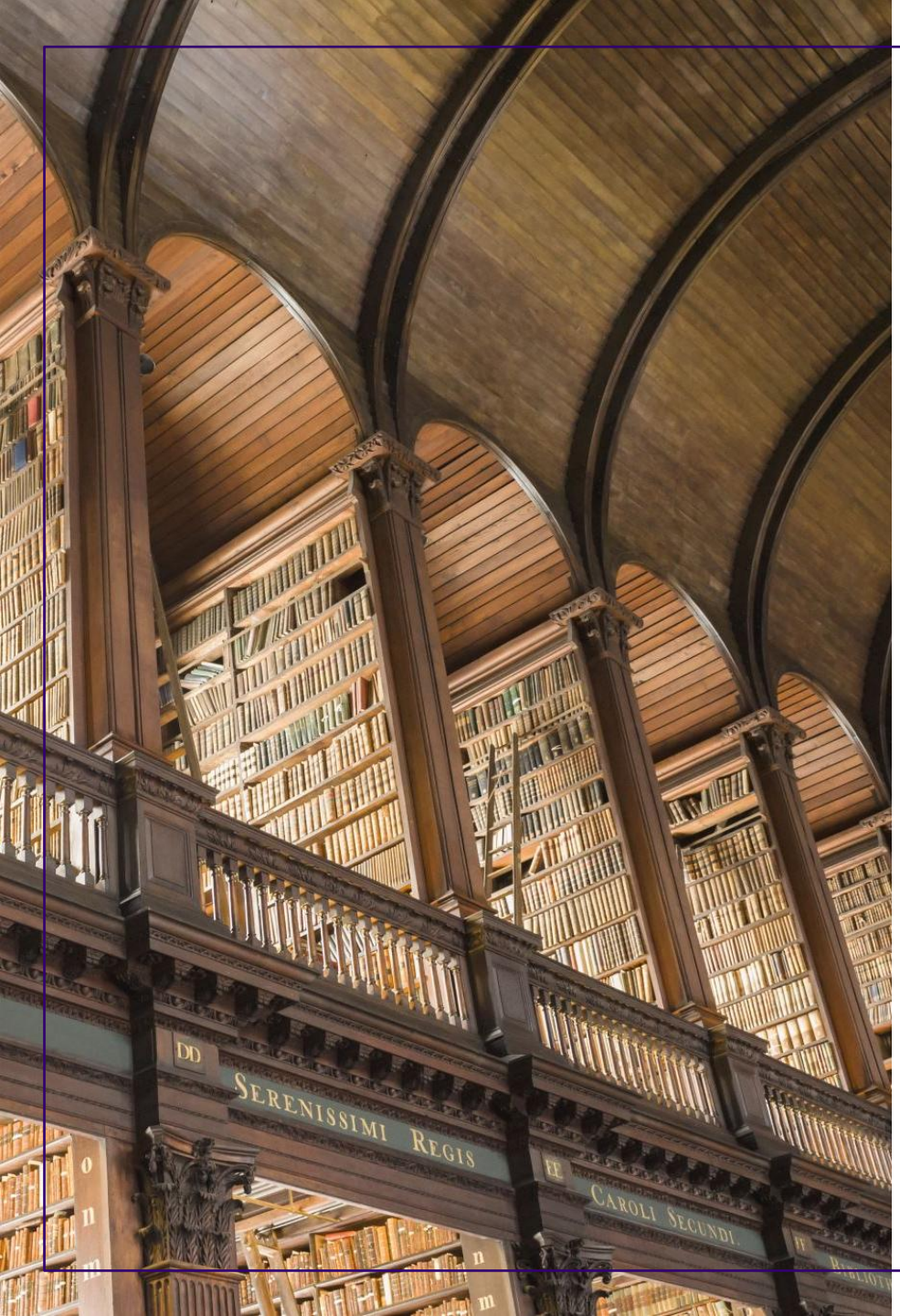
```
#ifndef PAIR_H_
#define PAIR_H_
#include <iostream>
#include <cstdlib>
template <typename Thing> class Pair {
 public:
  Pair() = default;  // initializes members
  Thing get_first() const { return first_; }
  Thing get_second() const { return second_; }
  void set_first(const Thing& copyme);
  void set_second(const Thing& copyme);
  void Swap();
 private:
  Thing first_, second_;
};
#include "Pair.cc"
#endif  // PAIR_H_
```

```
template <typename Thing>
void Pair<Thing>::set_first(const Thing& copyme) {
  first_ = copyme;
}

template <typename Thing>
void Pair<Thing>::Swap() {
  Thing tmp = first_;
  first_ = second_;
  second_ = tmp;
}
template <typename T>
std::ostream& operator<<(std::ostream& out, const
  Pair<T>& p) {
  return out << "Pair(" << p.get_first() << ", " <<
  p.get_second() << ")";
}
```

Class Templates – USING

```
#ifndef PAIR_H_
#define PAIR_H_
#include <iostream>
#include <cstdlib>
template <typename Thing> class Pair {
 public:
  Pair() = default;  // initializes members
  Thing get_first() const { return first_; }
  Thing get_second() const { return second_; }
  void set_first(const Thing& copyme);
  void set_second(const Thing& copyme);
  void Swap();
 private:
  Thing first_, second_;
};
#include "Pair.cc"
#endif  // PAIR_H_
```

```
#include "Pair.h"
using std::cout;
using std::endl;
using std::string;
int main(int argc, char** argv) {
  Pair<string> ps;  // this usage instantiates a
  class Pair<std::string>
  string x("foo"), y("bar");

  ps.set_first(x); // first_ = "foo", second_ = ""
  ps.set_second(y);//  second_ = "bar"
  ps.Swap();       // first_= "bar", second_= "foo"
  cout << ps << endl;  // nonmember overloaded
        operator<< invoked

  return EXIT_SUCCESS;
}
```



STANDARD TEMPLATE LIBRARY (STL)

STL Containers how-to

STL containers store by **value**, not by reference

- When you insert an object, the container makes a **copy**
- If the container needs to rearrange objects, it makes copies
  - *e.g.* if you sort a vector, it will make many, many copies
  - *e.g.* if you insert into a map, that may trigger several copies
- What if you don't want this (disabled copy constructor or copying is expensive)?
  - You can insert a wrapper object with a pointer to the object
    - We'll learn about these "smart pointers" soon

DEMOS

- Tracer class (for demonstration purposes)
- STL vector
- STL iterator
- STL algorithms

Tracer Class

Wrapper class for an `int value_`

- Holds unique `int id_` (increases from `0`)
- Default ctor (set unique `id_` for each instance), cctor, dtor, `op=`, `op<` defined
- `friend` function `operator<<` defined
- Private helper method `PrintID()` to return `"(id_,value_)"` as a string
- Class and member definitions can be found in `Tracer.h` and `Tracer.cc`

Useful for tracing behaviors of containers

- All methods print identifying messages
- Unique `id_` allows you to follow individual instances

```
class Tracer {
 public:
  Tracer();
  ~Tracer();
  Tracer(const Tracer &rhs);
  // Assignment operator.
  Tracer &operator=(const Tracer &rhs);
  // Less-than comparison operator.
  bool operator<(const Tracer &rhs) const;
  // << operator
  friend std::ostream &operator<<(std::ostream &out, const
 Tracer &rhs);
 private:
  // Return "id_ [history_]" as a string
  std::string Print(void) const;
  // This static (class member) tracks the next id_ to hand
 out.
  static int nextid_;
  const int id_;  // fixed id of this Tracer
  int value_;     // the most recent id
};
```

STL vector

- A generic, dynamically resizable array
- https://cplusplus.com/reference/vector/vector/
- Elements are stored in **contiguous** memory locations
  - Like a normal C array, or the ArrayList in Java!
  - Elements can be accessed using pointer arithmetic if you'd like
  - Random access is O(1) time
    - Pointer arithmetic, then access
- Adding/removing from the end is cheap (amortized constant time)
- Inserting/deleting from the middle or start is expensive (linear time)
  - Need to shift all of the elements in the array

```
#include <vector>
int main(int argc, char **argv) {
  Tracer a, b, c;
  vector<Tracer> vec;

  cout << "vec.push_back " << a << endl;
  vec.push_back(a);
  cout << "vec.push_back " << b << endl;
  vec.push_back(b);
  cout << "vec.push_back " << c << endl;
  vec.push_back(c);

  cout << "vec[0]" << endl;
  cout << vec[0] << endl;

  cout << "vec[2]" << endl;
  cout << vec[2] << endl;
  return EXIT_SUCCESS;
}
```

STL iterator

Each container class has an associated iterator class (*e.g.* `vector<int>::iterator`) used to iterate through elements of the container

https://cplusplus.com/reference/iterator/

- Iterator range is from `begin` up to `end` i.e., [`begin`, `end`)
  - `end` is one past the last container element!
- Some container iterators support more operations than others
  - All can be incremented (`++`), copied, copy-constructed
  - Some can be dereferenced on RHS (*e.g.* `x = *it;`)
  - Some can be dereferenced on LHS (*e.g.* `*it = x;`)
  - Some can be decremented (`--`)
  - Some support random access (`[]`, `+`, `-`, `+=`, `-=`, `<`, `>` operators)

```
#include <vector>

int main(int argc, char **argv) {
  Tracer a, b, c;
  vector<Tracer> vec;
  vec.push_back(a);
  vec.push_back(b);
  vec.push_back(c);

  cout << "Iterating:" << endl;
  vector<Tracer>::iterator it;
  for (it = vec.begin(); it < vec.end();
       it++) {
    cout << *it << endl;
  }
  cout << "Done iterating!" << endl;
  return EXIT_SUCCESS;
}
```

Type inference & Range for

The auto keyword can be used to infer types

- Simplifies your life if, for example, functions return complicated types
- The expression using auto must contain explicit initialization for it to work

```
// Inferred type
auto facts2 = Factors(12321);

// Compiler error here
auto facts3;
```

Syntactic sugar similar to Java's `foreach`

- declaration defines loop variable
- expression is an object representing a sequence
  - Strings, initializer lists, arrays with an explicit length defined, STL containers that support iterators

```
for ( declaration : expression ) {
  statements
}
// Prints out a string, one
// character per line
std::string str("hello");
for ( auto c : str ) {
  std::cout << c << std::endl;
}
```

STL *updated* iterator

Using newer language features such as ‘auto’ and the ‘range-for’ makes for cleaner code!

```
#include <vector>

int main(int argc, char **argv) {
  Tracer a, b, c;
  vector<Tracer> vec;
  vec.push_back(a);
  vec.push_back(b);
  vec.push_back(c);

  cout << "Iterating:" << endl;
  // "auto" is a C++11 feature not available
  // on older compilers
  for (auto &p : vec) {
    cout << p << endl;
  }
  cout << "Done iterating!" << endl;
  return EXIT_SUCCESS;
}
```

STL algorithms

A set of functions to be used on ranges of elements

- Range: any sequence that can be accessed through *iterators* or *pointers*, like arrays or some of the containers
- General form: `algorithm(begin, end, ...);`

Algorithms operate directly on range *elements* rather than the containers they live in

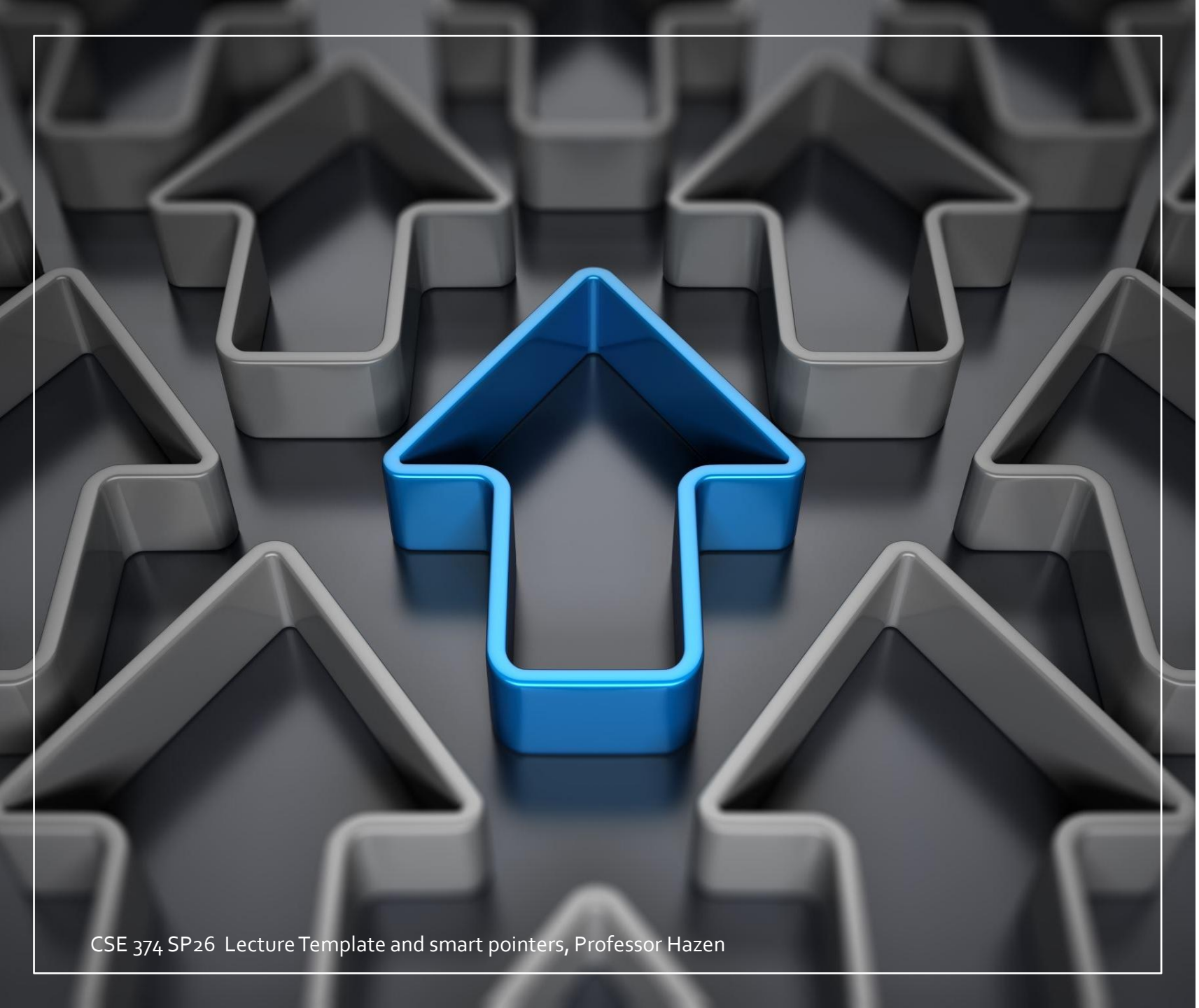
- Make use of elements' copy ctor, =, ==, !=, <
- Some do not modify elements
  - *e.g.* `find, count, for_each, min_element, binary_search`
- Some do modify elements
  - *e.g.* `sort, transform, copy, swap`

```
#include <vector>
#include <algorithm>
#include <cstdlib>
#include "Tracer.h"
using namespace std;

void PrintOut(const Tracer &p) {
  cout << "  printout: " << p << endl;
}
int main(int argc, char **argv) {
  Tracer a, b, c;
  vector<Tracer> vec;
  vec.push_back(c);
  vec.push_back(a);
  vec.push_back(b);
  sort(vec.begin(), vec.end());
  cout << "done sort!" << endl;
  for_each(vec.begin(), vec.end(), &PrintOut);
  return EXIT_SUCCESS;  }
```

Others

- `<list>` A generic doubly-linked list
- https://cplusplus.com/reference/list/list/
- Elements are **not** stored in contiguous memory locations
  - Does not support random access (*e.g.* cannot do list[5])
- Some operations are much more efficient than vectors
  - Constant time insertion, deletion anywhere in list
  - Can iterate forward or backwards
    - Backward: --
    - Forward: ++
- Has a built-in sort member function
  - Doesn't copy! Manipulates list structure instead of element values

- `<map>` One of C++'s *associative* containers: a key/value table, implemented as a search tree
- https://cplusplus.com/reference/map/
- General form: `map<key_type, value_type> name;`
- Keys must be *unique*
  - multimap allows duplicate keys
- Efficient lookup (O(log n)) and insertion (O(log n))
  - Access value via name[key]
    - If key doesn't exist in map, it is added to the map
- Elements are type pair<key_type, value_type> and are stored in *sorted* order (key is field first, value is field second)
- Key type must support less-than operator (<)



SMART POINTERS

What is a Smart Pointer?

A smart pointer is an **object** that stores a pointer to a heap-allocated object

- A smart pointer looks and behaves like a regular C++ pointer
  - By overloading `*`, `->`, `[]`, etc.
- These can help you manage memory
  - The smart pointer will delete the pointed-to object *at the right time* including invoking the object's destructor
    - When that is depends on what kind of smart pointer you use
  - With correct use of smart pointers, you no longer have to remember when to delete heap memory! (*If* it's owned by a smart pointer)

A ToyPtr class

We can use **<templates>** to make simple smart pointer

- A constructor accepts a pointer
- A destructor frees the pointer
- Overloaded * and → access the pointer

```
#ifndef TOYPTR_H_
#define TOYPTR_H_
template <typename T> class ToyPtr {
 public:
  explicit ToyPtr(T *ptr) : ptr_(ptr) { }

  // Destructor that deletes the ptr.
  ~ToyPtr() { delete ptr_; }
  // Implement the "*" operator
  T &operator*() { return *ptr_; }
  // Implement the "→" operator
  T *operator→() { return ptr_; }
 private:
  // The pointer itself.
  T *ptr_;
};
#endif  // TOYPTR_H_
```

Using a ToyPtr class

```
#include "./ToyPtr.h"
int main(int argc, char **argv) {
  // Create a dumb pointer.
  Point *leak = new Point;

  // Create a "smart" pointer.
  ToyPtr<Point> notleak(new Point);
  ToyPtr<Point> test(nullptr);

  std::cout << "*leak:" << *leak << std::endl;
  std::cout << "leak->x:" << leak->x << std::endl;
  std::cout << "*notleak:" << *notleak << std::endl;
  std::cout << "notleak->x:" << notleak->x <<
  std::endl;

  // Return, leaking "leak" but not "notleak".
  return EXIT_SUCCESS; }
```

```
#ifndef TOYPTR_H_
#define TOYPTR_H_
template <typename T> class ToyPtr {
 public:
  explicit ToyPtr(T *ptr) : ptr_(ptr) { }

  // Destructor that deletes the ptr.
  ~ToyPtr() { delete ptr_; }
  // Implement the "*" operator
  T &operator*() { return *ptr_; }
  // Implement the "->" operator
  T *operator->() { return ptr_; }
 private:
  // The pointer itself.
  T *ptr_;
};
#endif  // TOYPTR_H_
```

Breaking a ToyPtr class

It's a toy! (Here we would double delete, because we copy the pointer to x into y.)

Also can't handle arrays (need delete[])

Lots of ways to break it – luckily, we have non-toys available.

```
#include "./ToyPtr.h"

int main(int argc, char **argv) {

  // Create a "smart" pointer.
  ToyPtr<int> x(new int(5));
  ToyPtr<int> y = x;

  return EXIT_SUCCESS; }
```

```
#ifndef TOYPTR_H_
#define TOYPTR_H_
template <typename T> class ToyPtr {
 public:
  explicit ToyPtr(T *ptr) : ptr_(ptr) { }

  // Destructor that deletes the ptr.
  ~ToyPtr() { delete ptr_; }
  // Implement the "*" operator
  T &operator*() { return *ptr_; }
  // Implement the "->" operator
  T *operator->() { return ptr_; }
 private:
  // The pointer itself.
  T *ptr_;
};
#endif  // TOYPTR_H_
```

STD:SMART POINTERS

- After `-std=c++11` a suite of smart pointers was introduced
  - `std::unique_ptr` – sole owner of a pointer
  - `std::shared_ptr` – can have multiple owners of a pointer
  - `std::weak_ptr` – used with shared ptrs … (more later)
- These are implemented as templates: template parameter is the type that the "owned" pointer references (i.e., the T in pointer type T*)

unique_ptr

- A `unique_ptr` is the ***sole owner*** of a pointer
- Once we give a vanilla pointer a `unique_ptr`, we should stop using the original (non-smart) pointer
- Its destructor invokes delete on the owned pointer
  - Invoked when unique_ptr object is delete'd or falls out of scope via the unique_ptr destructor
- Guarantees uniqueness by disabling copy and assignment.

```
#include <iostream>
#include <memory>    // for std::unique_ptr
#include <cstdlib>

void Leaky() {
  int *x = new int(5);  // heap-allocated
  (*x)++;
  std::cout << *x << std::endl;
}  // never used delete, therefore leak
void NotLeaky() {
  std::unique_ptr<int> x(new int(5));
  (*x)++;
  std::cout << *x << std::endl;
}  // never used delete, but no leak

int main(int argc, char **argv) {
  Leaky();
  NotLeaky();
  return EXIT_SUCCESS; }
```

unique_ptr – use well

- A `unique_ptr` is the ***sole owner*** of a pointer
- Once we give a vanilla pointer a `unique_ptr`, we should stop using the original (non-smart) pointer
- Its destructor invokes delete on the owned pointer
  - Invoked when unique_ptr object is delete'd or falls out of scope via the unique_ptr destructor
- Guarantees uniqueness by disabling copy and assignment.

```
#include <iostream>
#include <memory>    // for std::unique_ptr
#include <cstdlib>

void Leaky() {
  int *x = new int(5);  // heap-allocated
  (*x)++;
  std::cout << *x << std::endl;
}  // never used delete, therefore leak
void NotLeaky() {
  std::unique_ptr<int> x(new int(5));
  (*x)++;
  std::cout << *x << std::endl;
}  // never used delete, but no leak

int main(int argc, char **argv) {
  Leaky();
  NotLeaky();
  return EXIT_SUCCESS; }
```

unique_ptr – why?

- A `unique_ptr` is the ***sole owner*** of a pointer
- If you have many potential exits out of a function, it's easy to forget to call delete on all of them
- unique_ptr will delete its pointer when it falls out of scope
- Thus, a unique_ptr also helps with *exception safety*

```
#include <iostream>
#include <memory>    // for std::unique_ptr
#include <cstdlib>

void NotLeaky() {
  std::unique_ptr<int> x(new int(5));
  (*x)++;
  // Potentially lots of code
  // including different return points
  std::cout << *x << std::endl;
}  // never used delete, but no leak

int main(int argc, char **argv) {
  NotLeaky();
  return EXIT_SUCCESS; }
```

unique_ptr – no copies!

- A `unique_ptr` is the ***sole owner*** of a pointer
- Disabled copy and assignment constructors

```
#include <memory> // for std::unique_ptr

using std::unique_ptr;
int main(int argc, char** argv) {
  unique_ptr<int> x(new int(5));

  // fail, no copy constructor
  unique_ptr<int> y(x);

  // succeed, z starts with NULL pointer
  unique_ptr<int> z;

  // fail, no assignment operator
  z = x;

  return EXIT_SUCCESS;
}
```

unique_ptr – pass ownership

- A `unique_ptr` is the ***sole owner*** of a pointer
- Use **reset**() and **release**() to transfer ownership
  - **release** returns the pointer, sets wrapped pointer to nullptr
  - **reset** delete's the current pointer and stores a new one

```
using std::cout;  using std::endl;
using std::unique_ptr;

int main(int argc, char **argv) {
  unique_ptr<int> x(new int(5));
  cout << "x: " << x.get() << endl;

  // y takes ownership, x abdicates it
  unique_ptr<int> y(x.release());
  cout << "x: " << x.get() << endl;
  cout << "y: " << y.get() << endl;

  unique_ptr<int> z(new int(10));

  // y transfers ownership to z.
  // z's old pointer is delete'd
  z.reset(y.release());
  return EXIT_SUCCESS;
}
```

unique_ptr – has array handling

- A `unique_ptr` is the ***sole owner*** of a pointer
- Calls `delete[]` upon destruction

```
#include <memory>
// for std::unique_ptr

using std::unique_ptr;

int main(int argc, char** argv) {
  // x is a unique_ptr that holds
  // the address of array of 5 ints
  unique_ptr<int[]> x(new int[5]);

  x[0] = 1;
  x[2] = 2;

  return EXIT_SUCCESS;
}
```

What if we want more than one copy of a pointer?

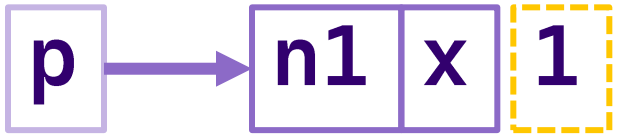
- `shared_ptr` is similar to `unique_ptr` but we allow shared objects to have multiple owners
- The copy/assign operators are not disabled and *increment* **reference counts** as needed
  - After a copy/assign, the two `shared_ptr` objects point to the same pointed-to object and the (shared) reference count is incremented by **`1`**
- When a `shared_ptr` is destroyed, the reference count is *decremented*
  - When the reference count hits **0**, we delete the pointed-to object!
- **Allows us to create complex linked structures (double-linked lists, graphs, etc.) at the cost of maintaining reference counts**

Reference Counting

Idea: associate a *reference count* with each object

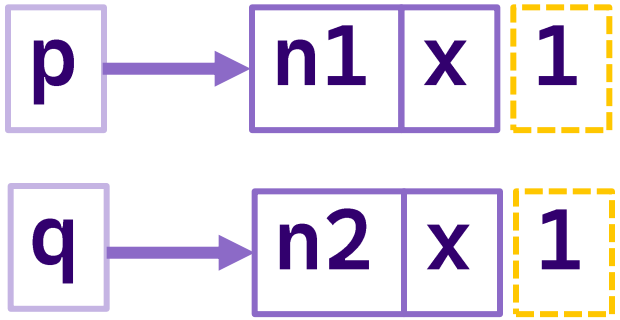
- Reference count holds number of **references** (pointers) to the object
- Adjusted whenever pointers are changed:
  - Increase by 1 each time we have a new pointer to an object
  - Decrease by 1 each time a pointer to an object is removed
- When reference counter decreased to 0, no more pointers to the object, so delete it (automatically)
- Used by C++ shared_ptr, not used in general for C++ memory management

Reference Counting (1)



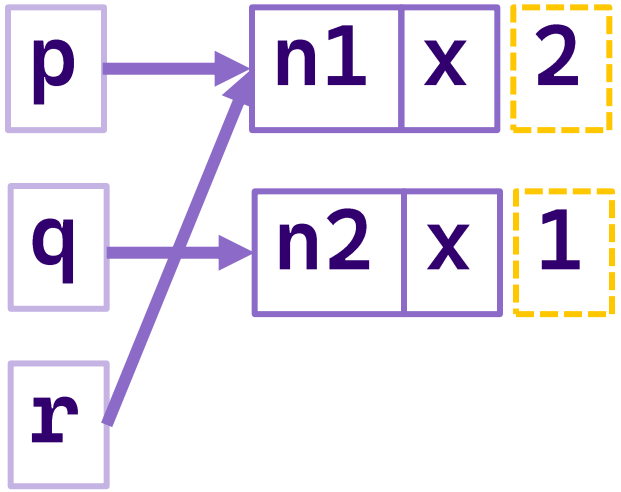
```
struct Node {
  int payload;  // node payload
  Node* next;  // next Node nullptr
};

Node * p = new Node();
Node * q = new Node();
Node * r = p;
q->next = new Node();
p = nullptr;
r = nullptr;
q = nullptr;
```

Reference Counting (2)



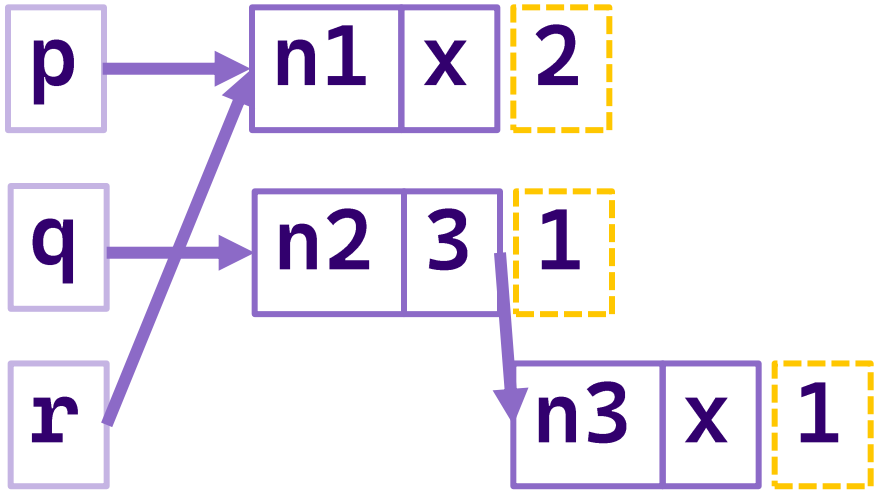
```
struct Node {
  int payload;  // node payload
  Node* next;  // next Node nullptr
};

Node * p = new Node();
Node * q = new Node();
Node * r = p;
q->next = new Node();
p = nullptr;
r = nullptr;
q = nullptr;
```

Reference Counting (3)



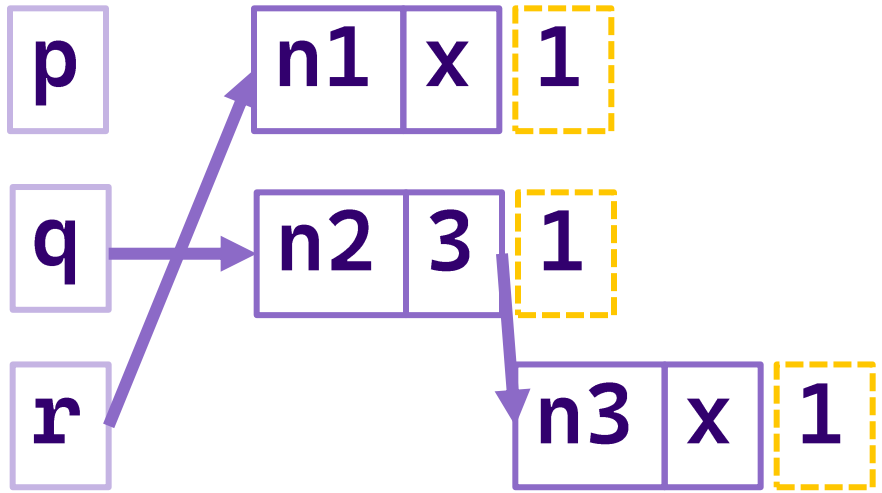
```
struct Node {
  int payload;  // node payload
  Node* next;  // next Node nullptr
};

Node * p = new Node();
Node * q = new Node();
Node * r = p;
q->next = new Node();
p = nullptr;
r = nullptr;
q = nullptr;
```

Reference Counting (4)



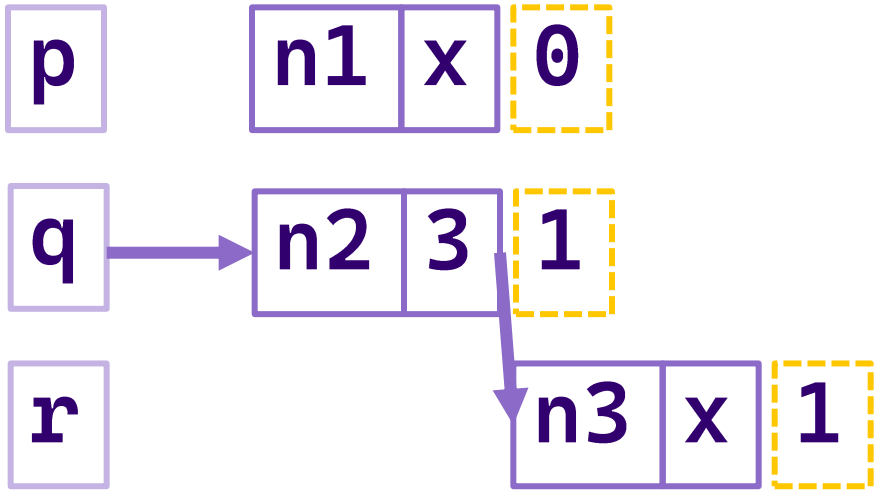

```
struct Node {
  int payload;  // node payload
  Node* next;  // next Node nullptr
};

Node * p = new Node();
Node * q = new Node();
Node * r = p;
q->next = new Node();
p = nullptr;
r = nullptr;
q = nullptr;
```

Reference Counting (5)



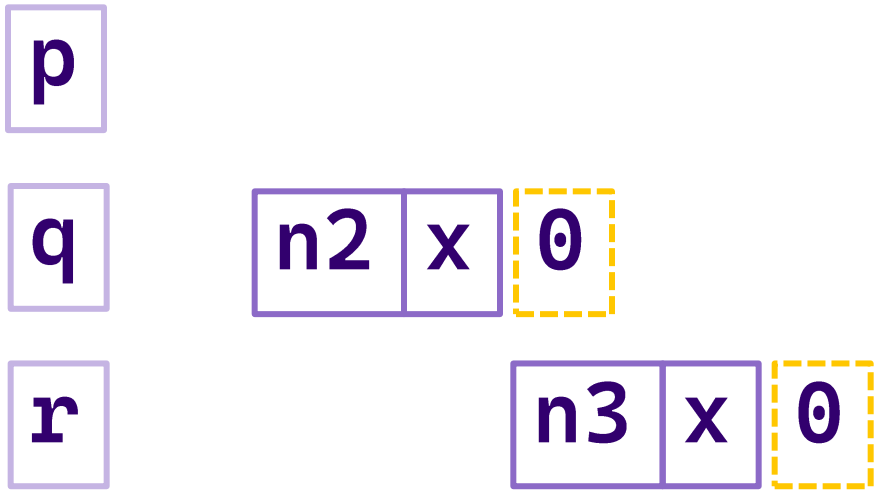

```
struct Node {
  int payload;  // node payload
  Node* next;  // next Node nullptr
};

Node * p = new Node();
Node * q = new Node();
Node * r = p;
q->next = new Node();
p = nullptr;
r = nullptr;
q = nullptr;
```

Reference Counting (6)




```
struct Node {
  int payload;  // node payload
  Node* next;  // next Node nullptr
};

Node * p = new Node();
Node * q = new Node();
Node * r = p;
q->next = new Node();
p = nullptr;
r = nullptr;
q = nullptr;
```

Reference Counting (7)




```
struct Node {
  int payload;  // node payload
  Node* next;  // next Node nullptr
};

Node * p = new Node();
Node * q = new Node();
Node * r = p;
q->next = new Node();
p = nullptr;
r = nullptr;
q = nullptr;  // cascades to also
              // delete n3
```

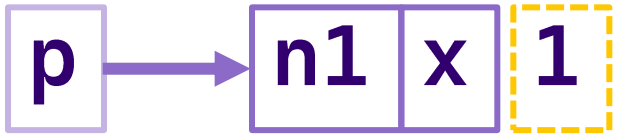
shared_ptr – example

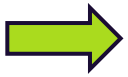
```
#include <memory>     // for std::shared_ptr

int main(int argc, char **argv) {
  std::shared_ptr<int> x(new int(10));   // ref count: 1
  {                 // temporary inner scope with local y (!)
    std::shared_ptr<int> y = x;          // ref count: 2
    std::cout << *y << std::endl;
  }                                      // y deleted

  std::cout << *x << std::endl;          // ref count: 1

  return EXIT_SUCCESS;
}                                        // ref count: 0
```

shared_ptr – can use with containers!

```
vector<std::shared_ptr<int>> vec;

vec.push_back(std::shared_ptr<int>(new int(9)));
vec.push_back(std::shared_ptr<int>(new int(5)));
vec.push_back(std::shared_ptr<int>(new int(7))); // don't copy objects

int& z = *vec[1];
std::cout << "z is: " << z << std::endl;

std::shared_ptr<int> copied = vec[1];  // works!
std::cout << "*copied: " << *copied << std::endl;
vec.pop_back();  // removes smart ptr & deallocate 7
```

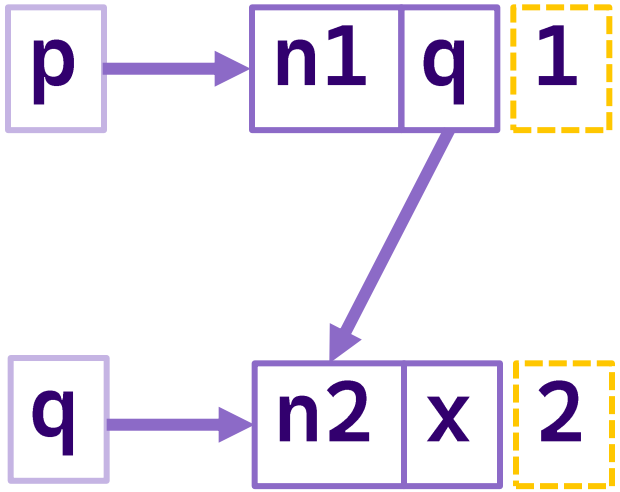
Reference Cycles (1)





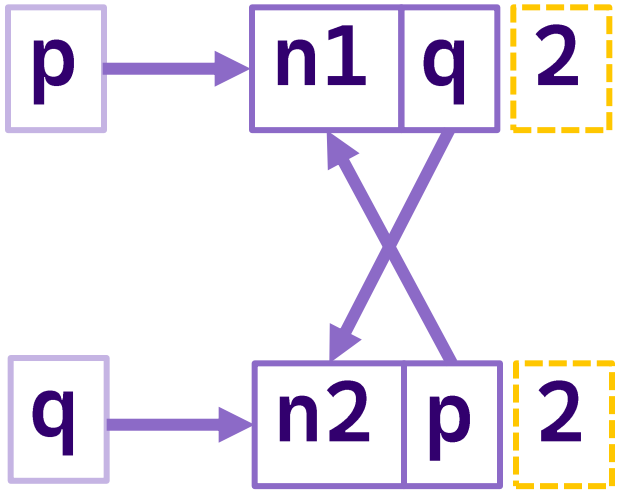
```
struct Node {
  int payload;  // node payload
  Node* next;  // next Node nullptr
};

Node * p = new Node();
Node * q = new Node();
p->next = q;
q->next = p;
p = nullptr;
q = nullptr;
```

Reference Cycles (2)



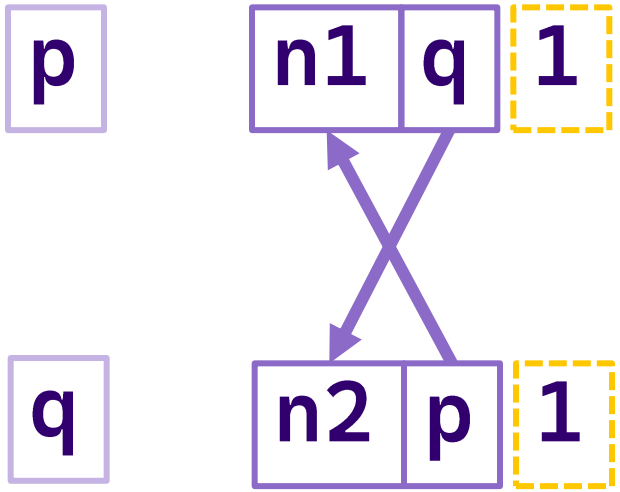

```
struct Node {
  int payload;  // node payload
  Node* next;  // next Node nullptr
};

Node * p = new Node();
Node * q = new Node();
p->next = q;
q->next = p;
p = nullptr;
q = nullptr;
```

Reference Cycles (3)




```
struct Node {
  int payload;  // node payload
  Node* next;  // next Node nullptr
};

Node * p = new Node();
Node * q = new Node();
p->next = q;
q->next = p;
p = nullptr;
q = nullptr;
```

Reference Cycles (4)




```
struct Node {
  int payload;  // node payload
  Node* next;  // next Node nullptr
};

Node * p = new Node();
Node * q = new Node();
p->next = q;
q->next = p;
p = nullptr;
q = nullptr;

// Memory Leak
```

Cycle of `shared_ptr`s

Head points to the first node,

Head->next points to the second node

Second-prev points to head-next

If we delete head, we have no way to get to nodes, but they aren't deleted because they each still have a positive reference count.

```
#include <cstdlib>
#include <memory>
using std::shared_ptr;

struct A {
  shared_ptr<A> next;
  shared_ptr<A> prev;
};

int main(int argc, char **argv) {
  shared_ptr<A> head(new A());
  head->next = shared_ptr<A>(new A());
  head->next->prev = head;
 // ref count of first node 2
  return EXIT_SUCCESS;
}
```

`weak_ptr`

`weak_ptr` is similar to a `shared_ptr` but doesn't affect the reference count

- Can *only* "point to" an object that is managed by a `shared_ptr`
- Not *really* a pointer – can't actually dereference unless you "get" its associated `shared_ptr`
- Because it doesn't influence the reference count, `weak_ptr`s can become "*dangling*"
  - Object referenced may have been `delete`'d
  - But you can check to see if the object still exists

Can be used to break our cycle problem!

Fixed cycle of `shared_ptr`s

Head points to the first node,

Head->next points to the second node

Second-prev points to head-next

Now, if we delete head, ref count goes to zero, so the first node is deleted, cascading to the second node.

```
#include <cstdlib>
#include <memory>
using std::shared_ptr;

struct A {
  shared_ptr<A> next;
  weak_ptr<A> prev;
};

int main(int argc, char **argv) {
  shared_ptr<A> head(new A());
  head->next = shared_ptr<A>(new A());
  head->next->prev = head;
 // ref count of first node 1
  return EXIT_SUCCESS;
}
```

Using a `weak_ptr`

```
#include <memory>                                  // for std::shared_ptr, std::weak_ptr
int main(int argc, char **argv) {
  std::weak_ptr<int> w;
  {                                                // temporary inner scope with local x
    std::shared_ptr<int> x;
    {                                              // temporary inner-inner scope with local y
      std::shared_ptr<int> y(new int(10));
      w = y;                                       // weak ref; ref count for “10” node is same
      x = w.lock();                                // get "promoted" shared_ptr, ref cnt = 2
      std::cout << *x << std::endl;
    }                                              // y deleted; ref count now 1
    std::cout << *x << std::endl;
  }                                                // x deleted; ref count now 0; mem freed
  std::shared_ptr<int> a = w.lock();               // nullptr
  std::cout << a << std::endl;                     // output is 0 (null)
  return EXIT_SUCCESS; }
```

Smart Pointer Functions

`std::unique_ptr U;`

- `U.get()` — Returns the raw pointer U is managing (⚠️ **Dangerous!**)
- `U.release()` — U stops managing its raw pointer and returns the raw pointer
- `U.reset(q)` — U cleans up its raw pointer and takes ownership of q

`std::shared_ptr S;`

- `make_shared<T>(args)` — Returns a `shared_ptr` pointer of a heap-allocated object
  `shared_ptr<int> p3 = make_shared<int>(42);`
- `S.use_count()` — Returns the reference count
- `S.unique()` — Returns true iff S.use_count() == 1

`std::weak_ptr W;`

- `W.lock()` — Constructs a shared pointer based off of W and returns it
- `W.use_count()` — Returns the reference count
- `W.expired()` — Returns true iff W is expired (W.use_count() == 0)

Why Not Smart Pointers?

- Smart pointers incur some overhead in both memory and CPU overhead
- Smart pointers are not always consistent with legacy code
- Smart pointers still don't know everything, you must be careful with what pointers you give it to manage.
  - Smart pointers can't tell if a pointer is on the heap or not.
  - Don't point smart pointers at the stack.
  - Still uses delete on default.
  - Smart pointers can't tell if you are re-using a raw pointer.
  - Circular references need extra work to detangle
  - Need for atomic operations can cause issues (lag) with multi-threaded operations

Memory Management Paradigms

Java

Tracing: Mark & Sweep

Mark all variables reachable from root objects, sweep remaining ones

Intermittent pauses while garbarge collection is done

Dangling references, GC can be unpredictable

C++ / smart_ptrs

Reference counting

Frees memory when reference count == 0

Added overhead to every allocation and deallocation

Issues: Cycles, overhead