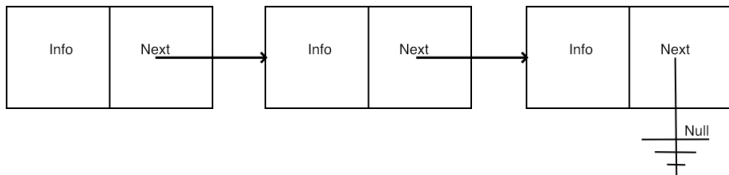# Smart Pointers

Victor Eijkhout, Susan Lindsey

Fall 2025
last formatted: November 6, 2025

1. Motivating application: linked list

- Used inside operating systems
- Model for complicated structures: trees, DAGs.

2. Recursive data structures

Naive code:

```
1 class Node {
2 private:
3   int value;
4   Node tail; // This does not work!
5   /* ... */
6 };
```

This does not work: would take infinite memory.

Indirect inclusion: only 'point' to the tail:

```
1 class Node {
2 private:
3   int value;
4   PointToNode tail;
5   /* ... */
6 };
```

3. Pointer types

- Smart pointers. You will see 'shared pointers'.
- There are 'unique pointers'. Those are tricky.
- Please don't use old-style C pointers,
  unless you become very advanced.

4. **Example: step 3: headers to include**

Using smart pointers requires at the top of your file:

```
1 #include <memory>
2 using std::shared_ptr;
3 using std::make_shared;
4
5 using std::unique_ptr;
6 using std::make_unique;
```

(unique pointers will not be discussed further here)

5. Example: step 1, we need a class

Simple class that stores one number:
Definition.
Example usage

```
1 // pointer/pointx.cpp
2 class HasX {
3 private:
4   double x;
5 public:
6   HasX( double x) : x(x) {};
7   auto value() { return x; };
8   void set(double xx) {
9     x = xx; };
10 };
```

```
1 // pointer/pointx.cpp
2 HasX xobj(5);
3 println("{}",xobj.value());
4 xobj.set(6);
5 println("{}",xobj.value());
```

6. Example: step 2, creating the pointer

Allocation of object and pointer to it in one:

```
auto X = make_shared<HasX>( /* args */ );

// or explicitly:

shared_ptr<HasX> X =
    make_shared<HasX>( /* constructor args */ );
```

7. Use of a shared pointer

Object vs pointed-object:

Code:

```cpp
// pointer/pointx.cpp
#include <memory>
using std::make_shared;

    /* ... */
    HasX xobj(5);
    println("{}",xobj.value());
    xobj.set(6);
    println("{}",xobj.value());

    auto xptr = make_shared<HasX>(5);
    println("{}",xptr->value());
    xptr->set(6);
    println("{}",xptr->value());
```

Output:

```
5
6
5
6
```

8. Example: step 4: in use

Set two pointers to the same object:

```
  Code:

1 // pointer/twopoint.cpp
2 auto xptr = make_shared<HasX>(5);
3 auto yptr = xptr;
4 cout << xptr->get() << '\n';
5 yptr->set(6);
6 cout << xptr->get() << '\n';
```

```
  Output:

1 5
2 6
```

What is the difference with

```
1 HasX xptr(5);
2 HasX yptr = xptr
3 cout << ...stuff...
```

?

9. **Pointer dereferencing**

Example: function

```
1 float distance_to_origin( Point p );
```

How do you apply that to a `shared_ptr<Point>`?
Use 'dereference' operator $*$:

```
1 shared_ptr<Point> p;
2 distance_to_origin( *p );
```

Exercise 1

Make a `DynRectangle` class, which is constructed from two shared-pointers-to-`Point` objects:

```
1 // pointer/dynrectangle.cpp
2 auto
3   origin  = make_shared<Point>(0,0),
4   fivetwo = make_shared<Point>(5,2);
5 DynRectangle lielow( origin,fivetwo );
```

Exercise 2

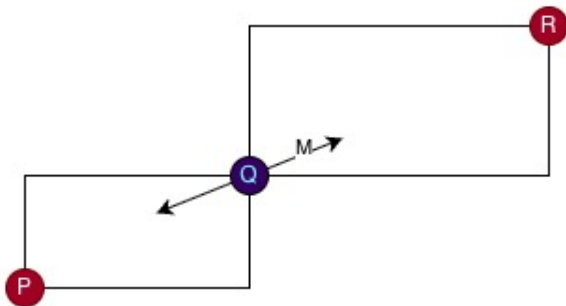Test this design: Calculate the area, scale the top-right point, and recalculate the area:

```
Code:

1 // pointer/dynrectangle.cpp
2 println( "Area: {}",lielow.area() );
3 /* ... */
4 println( "Area: {}",lielow.area() );
```

```
Output:

1 Area: 10
2 Area: 40
```

10. Illustration

Exercise 3

Make two *DynRectangle* objects so that the top-right corner of the first is the bottom-left corner of the other.

Now shift that point. Print out the two areas before and after to check correct behavior.

Use the *cxxopts* package to implement this syntax:

```
1 ./dynrectangle -p 1,1 -q 5,7 -r 10,12 -m 1,2
```

(referring to the previous picture)

Automatic memory management

11. Memory leaks

C has a 'memory leak' problem

```
1 // the variable `array' doesn't exist
2 {
3   // attach memory to `array':
4   double *array = new double[N];
5   // do something with array;
6   // forget to free
7 }
8 // the variable `array' does not exist anymore
9 // but the memory is still reserved.
```

The application 'is leaking memory'.
(even worse if you do this in a loop!)
Java/Python have 'garbage collection': runtime impact
C++ has the best solution: smart pointers with reference counting.

12. Illustration

We need a class with constructor and destructor tracing:

```cpp
// pointer/ptr1.cpp
class thing {
public:
  thing()  { cout << ".. calling constructor\n"; };
  ~thing() { cout << ".. calling destructor\n"; };
};
```

13. Constructor / destructor in action

Code:
```cpp
// pointer/ptr0.cpp
cout << "Outside\n";
{
  thing x;
  cout << "create done\n";
}
cout << "back outside\n";
```

Output:
```
Outside
.. calling
    ↪constructor
create done
.. calling destructor
back outside
```

14. **Illustration 1: pointer overwrite**

Let's create a pointer and overwrite it:

```
Code:

1 // pointer/ptr1.cpp
2 cout << "set pointer1"
3     << '\n';
4 auto thing_ptr1 =
5   make_shared<thing>();
6 cout << "overwrite pointer"
7     << '\n';
8 thing_ptr1 = nullptr;
```

```
Output:

1 set pointer1
2 .. calling
      ↪constructor
3 overwrite pointer
4 .. calling destructor
```

15. Illustration 2: pointer copy

```
Code:
1 // pointer/ptr2.cpp
2 cout << "set pointer2" << '\n';
3 auto thing_ptr2 =
4   make_shared<thing>();
5 cout << "set pointer3 by copy"
6     << '\n';
7 auto thing_ptr3 = thing_ptr2;
8 cout << "overwrite pointer2"
9     << '\n';
10 thing_ptr2 = nullptr;
11 cout << "overwrite pointer3"
12     << '\n';
13 thing_ptr3 = nullptr;
```
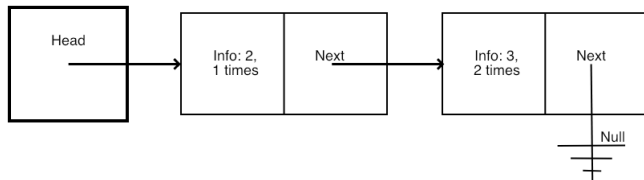
```
Output:
1 set pointer2
2 .. calling
     ↪constructor
3 set pointer3 by copy
4 overwrite pointer2
5 overwrite pointer3
6 .. calling destructor
```

Example: linked lists

16. Linked list

17. **Definition of List class**

A linked list has as its only member a pointer to a node:

```cpp
// tree/linkshared.cpp
class List {
private:
  shared_ptr<Node> head{nullptr};
public:
  List() {};
```

Initially null for empty list.

18. **Definition of Node class**

A node has information fields, and a link to another node:

```cpp
1  // tree/linkshared.cpp
2  class Node {
3  private:
4    int datavalue{0},datacount{0};
5    shared_ptr<Node> next{nullptr};
6  public:
7    Node() {};
8    Node(int value,shared_ptr<Node> next=nullptr)
9      : datavalue(value),datacount(1),next(next) {};
```

A Null pointer indicates the tail of the list.

19. **List methods**

List testing and modification.

```
1  List mylist;
2  cout << "Empty list has length: "
3      << mylist.length() << '\n';
4
5  mylist.insert(3);
6  cout << "After one insertion the length is: "
7      << mylist.length() << '\n';
8  if (mylist.contains_value(3))
9    cout << "Indeed: contains 3" << '\n';
```

20. **Print a list**

Auxiliary function so that we can trace what we are doing:

Print the list head.

```
1 // tree/linkshared.cpp
2 void List::print() {
3   cout << "List:";
4   if (head!=nullptr)
5     cout << " => "; head->print();
6   cout << '\n';
7 };
```

Print a node and its tail:

```
1 // tree/linkshared.cpp
2 void Node::print() {
3   cout << datavalue << ":" <<
       datacount;
4   if (has_next()) {
5     cout << ", "; next->print();
6   }
7 };
```

21. Recursive length computation

For the list:

```
// tree/linkshared.cpp
int List::length() {
  if (head==nullptr)
    return 0;
  else
    return head->length();
};
```

For a node:

```
// tree/linkshared.cpp
int Node::length() {
  if (!has_next())
    return 1;
  else
    return 1+next->length();
};
```

22. Iterative computation of the list length

Use a shared pointer to go down the list:

```
1 // tree/linkshared.cpp
2 int List::length_iterative() {
3   int count = 0;
4   if (head!=nullptr) {
5     auto current_node = head;
6     while (current_node->has_next()) {
7       current_node = current_node->nextnode(); count += 1;
8     }
9   }
10  return count;
11 };
```
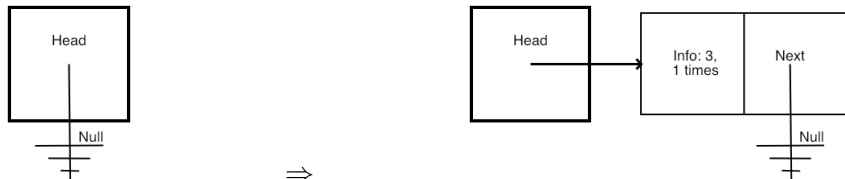
(Fun exercise: can do an iterative de-allocate of the list?)

Exercise 4

Test an empty list:

```
// catch/linktests.cpp
List mylist;
INFO( "Length of empty list" );
REQUIRE( mylist.length()==0 );
INFO( "Empty list has no headnode" );
REQUIRE_THROWS( mylist.headnode() );
```
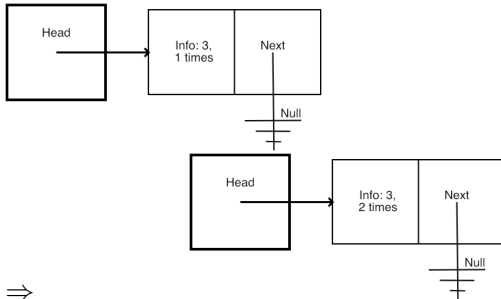
23. Creating the first list element

$\Rightarrow$

Exercise 5

Write the case of `Node::insert` that creates a node in an empty list.

```cpp
// catch/linktests.cpp
mylist.insert(3);
REQUIRE( mylist.length()==1 );
REQUIRE( mylist.contains_value(3) );
REQUIRE( not mylist.contains_value(4) );
```
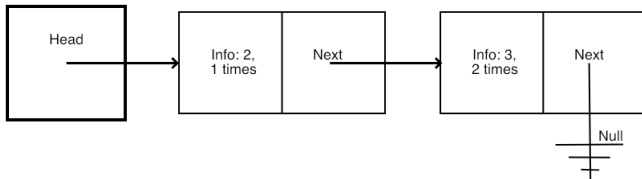
24. Elements that are already in the list

Exercise 6

Inserting a value that is already in the list means that the `count` value of a node needs to be increased. Update your `insert` method to make this code work:

```
1 // catch/linktests.cpp
2 REQUIRE_NOTHROW( mylist.insert(3) );
3 REQUIRE( mylist.length()==1 );
4 REQUIRE( mylist.contains_value(3) );
5 Node headnode;
6 REQUIRE_NOTHROW( headnode = mylist.headnode() );
7 REQUIRE( headnode.value()==3 );
8 REQUIRE( headnode.count()==2 );
```
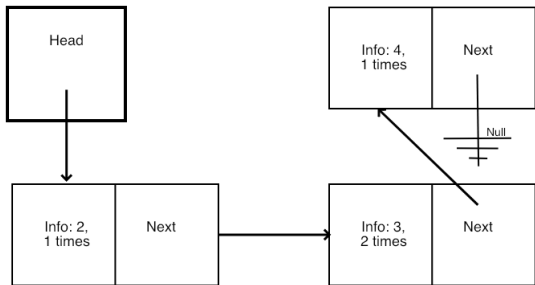
25. Element at the head

Exercise 7

One of the cases for inserting concerns an element that goes at the head. Update your `insert` method to get this to work:

```
1 // catch/linktests.cpp
2 REQUIRE_NOTHROW( mylist.insert(2) );
3 REQUIRE( mylist.length()==2 );
4 REQUIRE( mylist.contains_value(2) );
5 REQUIRE( mylist.contains_value(3) );
6
7 Node headnode;
8 INFO( "The head node should have 2, not 3" );
9 REQUIRE_NOTHROW( headnode = mylist.headnode() );
10 REQUIRE( headnode.value()==2 );
11 REQUIRE( headnode.count()==1 );
```
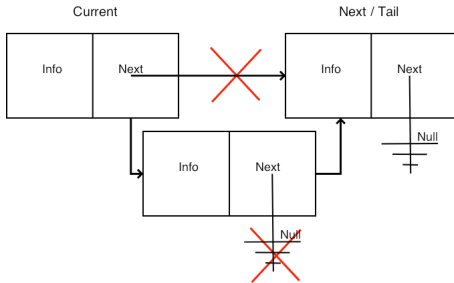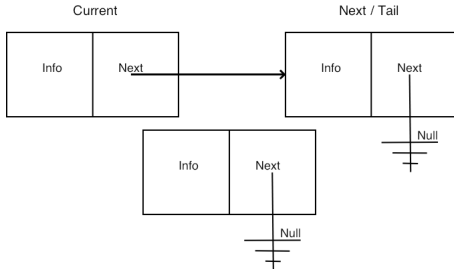
26. Element at the tail

Exercise 8

If an item goes at the end of the list:

```cpp
// catch/linktests.cpp
REQUIRE_NOTHROW( mylist.insert(6) );
REQUIRE( mylist.length()==3 );
REQUIRE( mylist.contains_value(6) );
```

27. Insertion

Exercise 9

Update your insert routine to deal with elements that need to go somewhere in the middle.

```cpp
// catch/linktests.cpp
REQUIRE_NOTHROW( mylist.insert(4) );
REQUIRE( mylist.length()==4 );
REQUIRE( mylist.contains_value(4) );
REQUIRE( mylist.contains_value(3) );
```