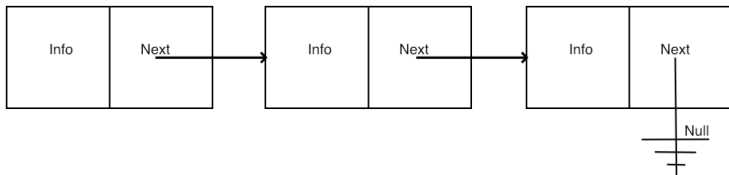# Smart Pointers

Victor Eijkhout, Susan Lindsey

Fall 2024
last formatted: October 18, 2024

1. Motivating application: linked list

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

2. Recursive data structures

Naive code:

```cpp
class Node {
private:
  int value;
  Node tail;
  /* ... */
};
```

This does not work: would take infinite memory.

Indirect inclusion: only 'point' to the tail:

```cpp
class Node {
private:
  int value;
  PointToNode tail;
  /* ... */
};
```

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 1, we need a class

Simple class that stores one number:
Definition:

```
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 };
```

Example usage

```
1 // pointer/pointx.cpp
2 HasX xobj(5);
3 cout << xobj.value() << '\n';
4 xobj.set(6);
5 cout << xobj.value() << '\n';
```

5. 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 */ );
```

6. Use of a shared pointer

Object vs pointed-object:

```
Code:
1 // pointer/pointx.cpp
2 #include <memory>
3 using std::make_shared;
4
5    /* ... */
6    HasX xobj(5);
7    cout << xobj.value() << '\n';
8    xobj.set(6);
9    cout << xobj.value() << '\n';
10
11   auto xptr = make_shared<HasX>(5);
12   cout << xptr->value() << '\n';
13   xptr->set(6);
14   cout << xptr->value() << '\n';
```

```
Output:

5
6
5
6
```

7. Example: step 3: headers to include

Using smart pointers requires at the top of your file:

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

using std::unique_ptr;
using std::make_unique;
```

(unique pointers will not be discussed further here)

8. Example: step 4: in use

Set two pointers to the same object:

```
Code:

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

```
Output:

5
6
```

What is the difference with

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

?

9. Pointer dereferencing

Example: function

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

How do you apply that to a shared_ptr<Point>?

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

10. **Null pointer**

Initialize smart pointer to null pointer; test on null value:

```
shared_ptr<Foo> foo_ptr = nullptr;
// stuff
if (foo_ptr!=nullptr)
  foo_ptr->do_something();
```

Exercise 1

With this code given:

```
Code:

1  // pointer/dynrectangle.cpp
2  float dx( Point other ) {
3    return other.x-x; };
4  /* ... */
5  // main, with objects
6  Point
7    oneone(1,1), fivetwo(5,2);
8  float dx = oneone.dx(fivetwo);
9  /* ... */
10 // main, with pointers
11 auto
12   oneonep = make_shared<Point>(1,1),
13   fivetwop = make_shared<Point>(5,2);
```

```
Output:

dx: 4
dx: 4
```

compute the `dx` between the `oneonep` & `fivetwop`.

You can base this off the file dynrectangle.cpp in the repository

Exercise 2

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 3

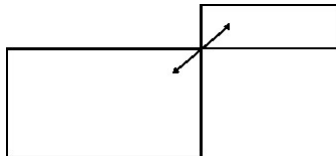Test this design: Calculate the area, scale the top-right point, and recalculate the area:

```
Code:

1 // pointer/dynrectangle.cpp
2 cout << "Area: " <<
      lielow.area() << '\n';
3 /* ... */
4 cout << "Area: " <<
      lielow.area() << '\n';
```

```
Output:

Area: 10
Area: 40
```

11. **For the next exercise**

Exercise 4

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.

Automatic memory management

12. Memory leaks

C has a 'memory leak' problem

```
// the variable `array' doesn't exist
for ( /* lots of iterations */ ) {
  // attach memory to `array':
  double *array = new double[N];
  // do something with array;
  // forget to free
}
// the variable `array' does not exist anymore
// but the memory is still reserved.
```

The application 'is leaking memory'.
Java/Python have 'garbage collection': runtime impact
C++ has the best solution: smart pointers with reference counting.

13. **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"; };
};
```

14. Constructor / destructor in action

```
Code:

1 // pointer/ptr0.cpp
2 cout << "Outside\n";
3 {
4   thing x;
5   cout << "create done\n";
6 }
7 cout << "back outside\n";
```

```
Output:

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

15. **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:

set pointer1
.. calling
    ↪constructor
overwrite pointer
.. calling destructor
```

16. Illustration 2: pointer copy

Code:

```cpp
// pointer/ptr2.cpp
cout << "set pointer2" << '\n';
auto thing_ptr2 =
  make_shared<thing>();
cout << "set pointer3 by copy"
     << '\n';
auto thing_ptr3 = thing_ptr2;
cout << "overwrite pointer2"
     << '\n';
thing_ptr2 = nullptr;
cout << "overwrite pointer3"
     << '\n';
thing_ptr3 = nullptr;
```
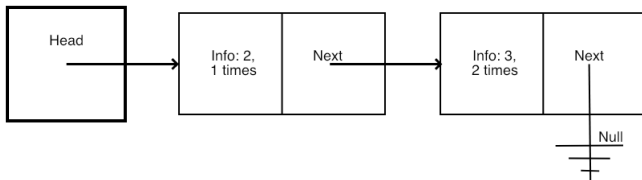
Output:

```
set pointer2
.. calling
    ↪constructor
set pointer3 by copy
overwrite pointer2
overwrite pointer3
.. calling destructor
```

17. Reference counting

- The object counts how many pointers there are:
- 'reference counting'
- A pointed-to object is deallocated if no one points to it.

Example: linked lists

18. **Linked list**

You can base this off the file `linkshared.cpp` in the repository

19. **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.

20. Definition of Node class

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

```cpp
// tree/linkshared.cpp
class Node {
private:
  int datavalue{0},datacount{0};
  shared_ptr<Node> next{nullptr};
public:
  Node() {};
  Node(int value,shared_ptr<Node> next=nullptr)
    : datavalue(value),datacount(1),next(next) {};
```

A Null pointer indicates the tail of the list.

21. List methods

List testing and modification.

```
List mylist;
cout << "Empty list has length: "
     << mylist.length() << '\n';

mylist.insert(3);
cout << "After one insertion the length is: "
     << mylist.length() << '\n';
if (mylist.contains_value(3))
  cout << "Indeed: contains 3" << '\n';
```

22. Recursive functions

- List structure is recursive
- Algorithms are naturally formulated recursively.

23. Recursive length computation

For the list:

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

For a node:

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

24. Iterative functions

- Recursive functions may have performance problems
- Iterative formulation possible

25. Iterative computation of the list length

Use a shared pointer to go down the list:

```cpp
// tree/linkshared.cpp
int List::length_iterative() {
  int count = 0;
  if (head!=nullptr) {
    auto current_node = head;
    while (current_node->has_next()) {
      current_node = current_node->nextnode(); count += 1;
    }
  }
  return count;
};
```

(Fun exercise: can do an iterative de-allocate of the list?)

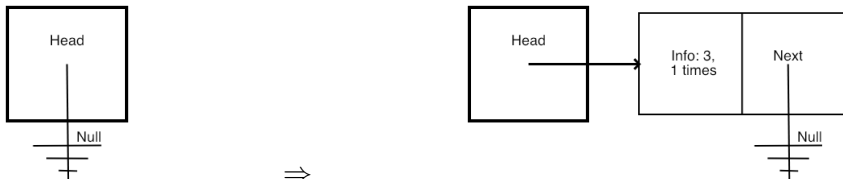# 26. Print a list

Auxiliary function so that we can trace what we are doing.
Print the list head: Print a node and its tail:

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

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

# 27. Creating the first list element

Exercise 5

Next write the case of *Node::insert* that handles the empty list.
You also need a method *List::contains* that tests if an item if in
the list.

```cpp
// tree/linkshared.cpp
mylist.insert(3);
cout << "After inserting 3 the length is: "
    << mylist.length() << '\n';
if (mylist.contains_value(3))
  cout << "Indeed: contains 3" << '\n';
else
  cout << "Hm. Should contain 3" << '\n';
if (mylist.contains_value(4))
  cout << "Hm. Should not contain 4" << '\n';
else
  cout << "Indeed: does not contain 4" << '\n';
cout << '\n';
```

# 28. Elements that are already in the list
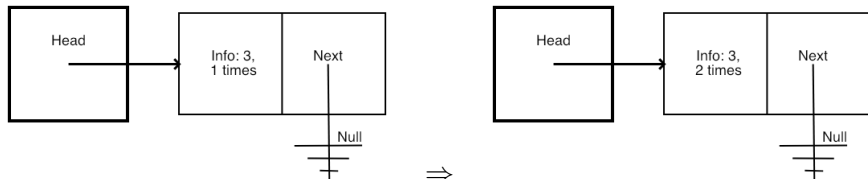
Exercise 6

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:

```cpp
// tree/linkshared.cpp
mylist.insert(3);
cout << "Inserting the same item gives length: "
    << mylist.length() << '\n';
if (mylist.contains_value(3)) {
  cout << "Indeed: contains 3" << '\n';
  auto headnode = mylist.headnode();
  cout << "head node has value " << headnode->value()
      << " and count " << headnode->count() << '\n';
} else
  cout << "Hm. Should contain 3" << '\n';
cout << '\n';
```

29. Element at the head
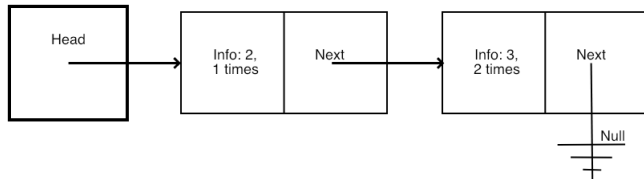
Exercise 7

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

```cpp
// tree/linkshared.cpp
mylist.insert(2);
cout << "Inserting 2 goes at the head;\nnow the length is: "
     << mylist.length() << '\n';
if (mylist.contains_value(2))
  cout << "Indeed: contains 2" << '\n';
else
  cout << "Hm. Should contain 2" << '\n';
if (mylist.contains_value(3))
  cout << "Indeed: contains 3" << '\n';
else
  cout << "Hm. Should contain 3" << '\n';
cout << '\n';
```
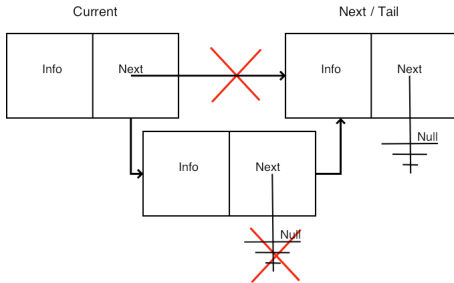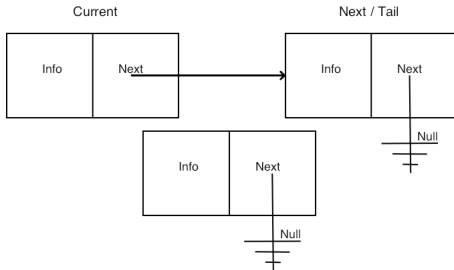
30. Element at the tail
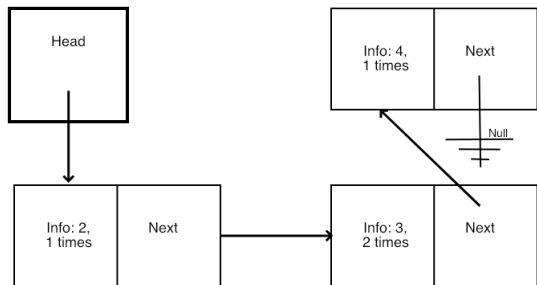
Exercise 8

If an item goes at the end of the list:

```
1 // tree/linkshared.cpp
2 mylist.insert(6);
3 cout << "Inserting 6 goes at the tail;\nnow the length is: "
4     << mylist.length()
5     << '\n';
6 if (mylist.contains_value(6))
7   cout << "Indeed: contains 6" << '\n';
8 else
9   cout << "Hm. Should contain 6" << '\n';
10 if (mylist.contains_value(3))
11   cout << "Indeed: contains 3" << '\n';
12 else
13   cout << "Hm. Should contain 3" << '\n';
14 cout << '\n';
```

31. Insertion

Exercise 9

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

```cpp
// tree/linkshared.cpp
mylist.insert(4);
cout << "Inserting 4 goes in the middle;\nnow the length is: "
    << mylist.length()
    << '\n';
if (mylist.contains_value(4))
  cout << "Indeed: contains 4" << '\n';
else
  cout << "Hm. Should contain 4" << '\n';
if (mylist.contains_value(3))
  cout << "Indeed: contains 3" << '\n';
else
  cout << "Hm. Should contain 3" << '\n';
cout << '\n';
```

32. Linked list exercise

Write a program that constructs a linked list where the elements are sorted in increasing numerical order.

Your program should accept a sequence of numbers from interactive input, and after each number print the list for as far as it has been constructed. Print the list on a single line, with elements separated by commas.

An input value of zero signals the end of input; this number is not added to the list.