

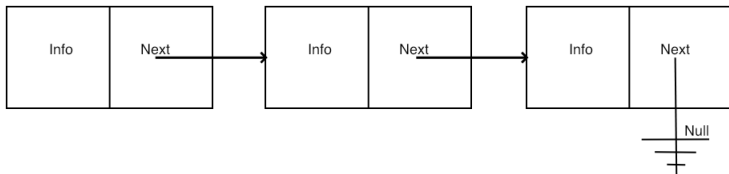
# Smart Pointers

Victor Eijkhout, Susan Lindsey

Fall 2023

last formatted: October 9, 2023

# 1. Motivating application: linked list



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

## 2. Recursive data structures

Naive code:

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

This does not work: would take infinite memory.

Indirect inclusion: only 'point' to the tail:

```
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:

```
// pointer/pointx.cpp
class HasX {
private:
    double x;
public:
    HasX( double x) : x(x) {};
    auto value() { return x; };
    void set(double xx) {
        x = xx; };
};
```

Example usage

```
// pointer/pointx.cpp
HasX xobj(5);
cout << xobj.value() <<
'\n';
xobj.set(6);
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 =
12     make_shared<HasX>(5);
13 cout << xptr->value() << '\n';
14 xptr->set(6);
15 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

Why do we use pointers?

Pointers make it possible for two variables to own 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 );
```

# Exercise 1

With this code given:

Code:

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

Output:

```
dx: 4
dx: 4
```

compute the  $dx$  between the 'pointered' *Points*.

## Exercise 2

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

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

## Exercise 3

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

Code:

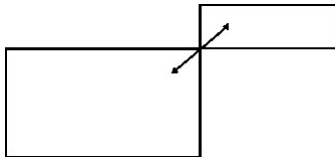
```
1 // pointer/dynrectangle.cpp
2 cout << "Area: " <<
    lielow.area() << '\n';
3 /* ... */
4 cout << "Area: " <<
    lielow.area() << '\n';
```

Output:

```
Area: 10
Area: 40
```

*You can base this off the file `pointrectangle.cxx` in the repository*

## 10. 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**



# 11. Memory leaks

C has a 'memory leak' problem

```
// the variable `array' doesn't exist
{
    // 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'.

(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:

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

## 13. Show 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
```

## 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:

```
set pointer1
.. calling constructor
overwrite pointer
.. 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;
```

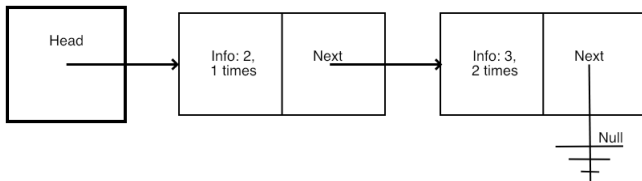
Output:

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

- 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

## 16. Linked list



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

## 17. Definition of List class

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

```
// 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:

```
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) {};
10    int value() {
11        return datavalue; };
12    auto nextnode() {
13        return next; };
```

A Null pointer indicates the tail of the list.

## 19. 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';
```

## 20. Print a list

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

Print the list head:

```
// tree/linkshared.cpp
void print() {
    cout << "List:";
    if (has_list())
        cout << " => ";
    head->print();
    cout << '\n';
};
```

Print a node and its tail:

```
// tree/linkshared.cpp
void print() {
    cout << datavalue << ":"
        << datacount;
    if (has_next()) {
        cout << ", ";
        next->print();
    }
};
```

## 21. Recursive length computation

For the list:

```
// tree/linkunique.cpp
int recursive_length() {
    if (!has_list())
        return 0;
    else
        return head->listlength();
};
```

For a node:

```
// tree/linkunique.cpp
int listlength_recursive() {
    if (!has_next()) return 1;
    else return 1+next->listlength_recursive();
};
```

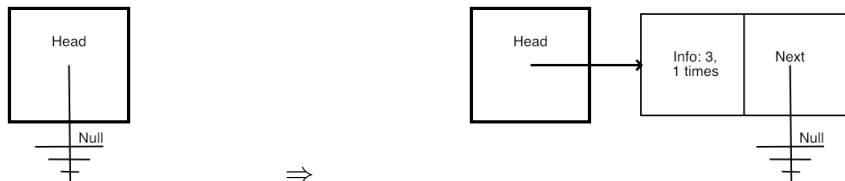
## 22. Iterative computation of the list length

Use a shared pointer to go down the list:

```
// tree/linkshared.cpp
int length_iterative() {
    int count = 0;
    if (has_list()) {
        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?)

## 23. Creating the first list element

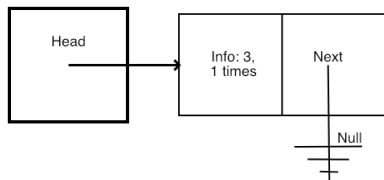


## Exercise 5

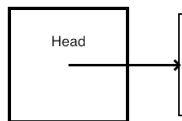
Next write the case of *Node::insert* that handles the empty list. You also need a method *List::contains* that tests if an item is in the list.

```
// 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';
```

## 24. Elements that are already in the list



$\Rightarrow$



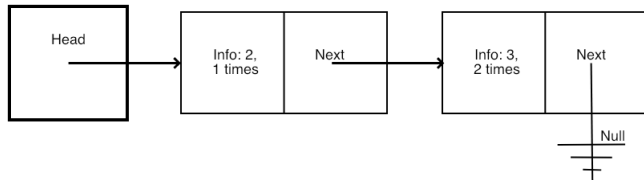


## Exercise 6

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:

```
// 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';
```

## 25. Element at the head

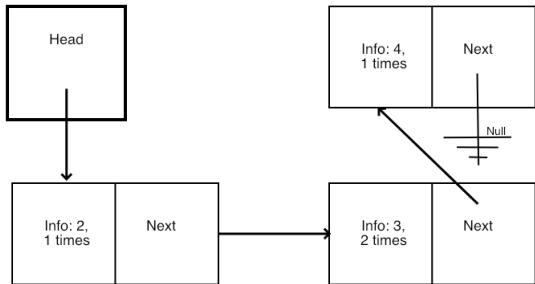


## Exercise 7

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

```
// 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';
```

## 26. Element at the tail

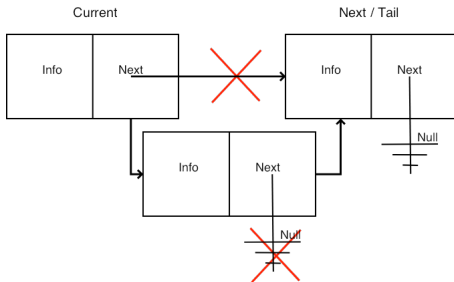
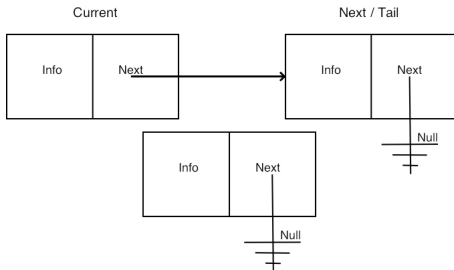


## Exercise 8

If an item goes at the end of the list:

```
// tree/linkshared.cpp
mylist.insert(6);
cout << "Inserting 6 goes at the tail;\nnow the length is: "
    << mylist.length()
    << '\n';
if (mylist.contains_value(6))
    cout << "Indeed: contains 6" << '\n';
else
    cout << "Hm. Should contain 6" << '\n';
if (mylist.contains_value(3))
    cout << "Indeed: contains 3" << '\n';
else
    cout << "Hm. Should contain 3" << '\n';
cout << '\n';
```

# 27. Insertion



## Exercise 9

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

```
// 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';
```

## 28. 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.



# Turn it in!

- If you have compiled your program, do:  
`coe_link yourprogram.cxx`  
where 'yourprogram.cxx' stands for the name of your source file.
- Is it reporting that your program is correct? If so, do:  
`coe_link -s yourprogram.cc`  
where the -s flag stands for 'submit'.
- If you don't manage to get your code working correctly, you can submit as incomplete with  
`coe_link -i yourprogram.cc`