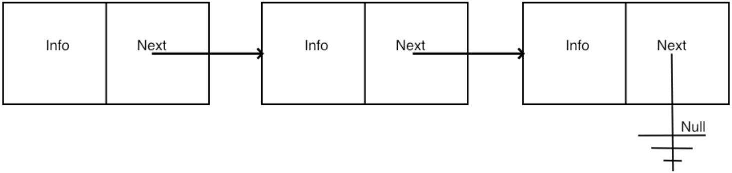
Smart Pointers

Victor Eijkhout, Susan Lindsey

Fall 2023

`last formatted: October 13, 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:

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

    /* ... */
    HasX xobj(5);
    cout << xobj.value() << '\n';
    xobj.set(6);
    cout << xobj.value() << '\n';

    auto xptr =
     make_shared<HasX>(5);
    cout << xptr->value() << '\n';
    xptr->set(6);
    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:

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

```
// pointer/dynrectangle.cpp
  float dx( Point other ) {
    return other.x-x; };
    /* ... */
    // main, with objects
    Point
      oneone(1,1), fivetwo(5,2);
    float dx = oneone.dx(fivetwo);
    /* ... */
    // main, with pointers
    auto
      oneonep  = make_shared<Point>(1,1),
      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:

```
// 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:

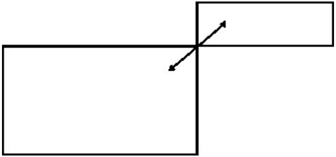
Code:

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

13. 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"; };
};
```

14. Show constructor / destructor in action

Code:

```
// 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
```

15. Illustration 1: pointer overwrite

Let's create a pointer and overwrite it:

Code:

```
// pointer/ptr1.cpp
  cout << "set pointer1"
       << '\n';
  auto thing_ptr1 =
    make_shared<thing>();
  cout << "overwrite pointer"
       << '\n';
  thing_ptr1 = nullptr;
```

Output:

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

16. Illustration 2: pointer copy

Code:

```
// 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;
```

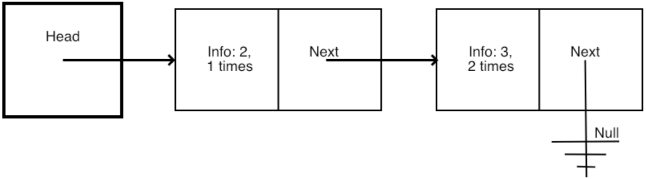
Output:

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

- 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

17. Linked list



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

18. 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.

19. Definition of Node class

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

```
// 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.

20. 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';
```

21. Recursive functions

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

22. Recursive length computation

For the list:

```
// tree/linkshared.cpp
int List::length() {
  int count = 0;
  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();
};
```

23. Iterative functions

- Recursive functions may have performance problems
- Iterative formulation possible

24. Iterative computation of the list length

Use a shared pointer to go down the list:

```
// 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?)

25. Print a list

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

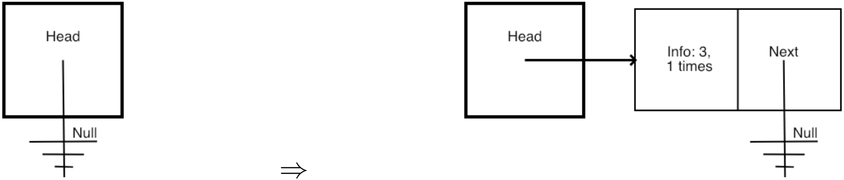
Print the list head:

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

Print a node and its tail:

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

26. Creating the first list element

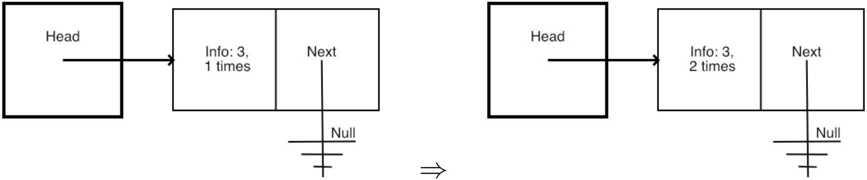


Exercise 5

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.

```
// 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';
```

27. Elements that are already in the list

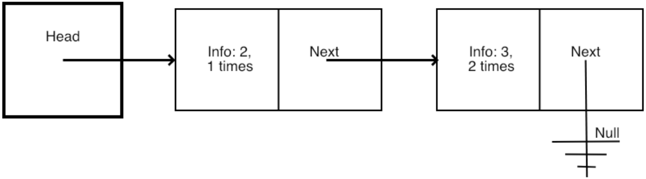


Exercise 6

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:

```
// 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';
```

28. Element at the head

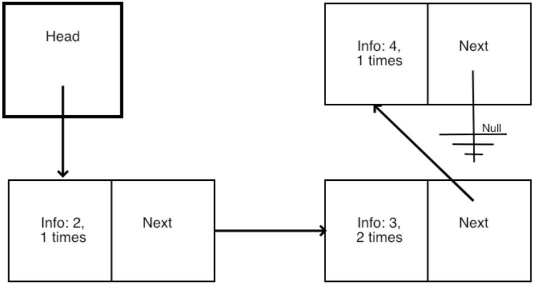


Exercise 7

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';
```

29. Element at the tail

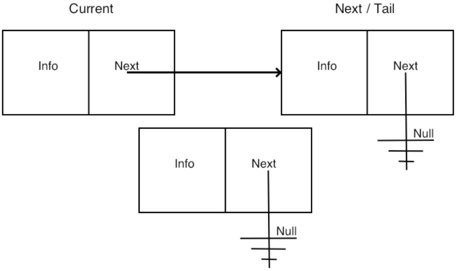


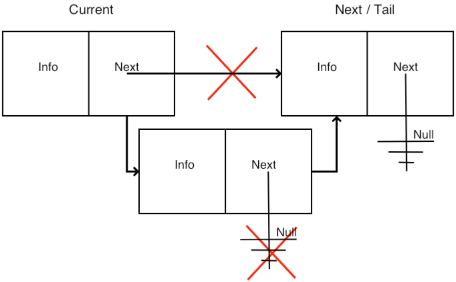
Exercise 8

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';
```

30. 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';
```

31. 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.