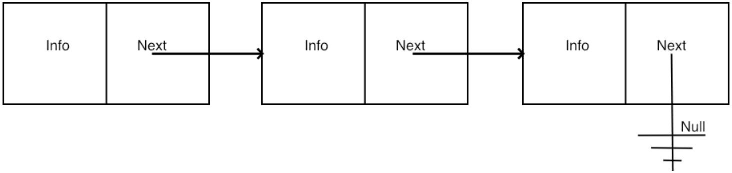
Smart Pointers

Victor Eijkhout, Susan Lindsey

Fall 2022

last formatted: October 11, 2022

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:

```
class HasX {
private:
  double x;
public:
  HasX( double x) : x(x) {};
  auto get() { return x; };
  void set(double xx) { x = xx; };
};
```

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. 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)

7. Example: step 4: in use

Why do we use pointers?

Pointers make it possible for two variables to own the same object.

```
Code:

auto xptr = make_shared<HasX>(5);
auto yptr = xptr;
cout << xptr->get() << '\n';
yptr->set(6);
cout << xptr->get() << '\n';
```

```
Output
[pointer] twopoint:

5
6
```

What is the difference with

```
HasX five(5);
HasX v1 = five;
HasX v2 = five;
```

?

Exercise 1

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

```
auto
  origin  = make_shared<Point>(0,0),
  fivetwo = make_shared<Point>(5,2);
DynRectangle lielow( origin,fivetwo );
```

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

```
Code:

cout << "Area: " <<  lielow.area() <<
    '\n';
/* ... */
// scale the 'fivetwo' point by two
cout << "Area: " <<  lielow.area() <<
    '\n';
```

```
Output
[pointer] dynrect:

Area: 10
Area: 40
```

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

Automatic memory management

8. 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
}
// 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.

9. Illustration

We need a class with constructor and destructor tracing:

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

10. Show constructor / destructor in action

Code:

```
cout << "Outside\n";
{
  thing x;
  cout << "create done\n";
}
cout << "back outputside\n";
```

Output
[pointer] ptr0:

```
Outside
.. calling
     constructor
create done
.. calling destructor
back outputside
```

11. Illustration 1: pointer overwrite

Let’s create a pointer and overwrite it:

```
Code:

cout << "set pointer1"
     << ’\n’;
auto thing_ptr1 =
  make_shared<thing>();
cout << "overwrite pointer"
     << ’\n’;
thing_ptr1 = nullptr;
```

```
Output
[pointer] ptr1:

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

12. Illustration 2: pointer copy

```
Code:

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;
```

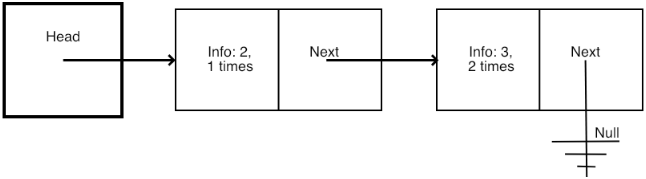
```
Output
[pointer] ptr2:

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

13. Linked list



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

14. Definition of List class

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

```
class List {
private:
  shared_ptr<Node> head{nullptr};
public:
  List() {};
```

Initially null for empty list.

15. Definition of Node class

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

```
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) {};
  int value() {
    return datavalue; };
  auto nextnode() {
    return next; };
```

A Null pointer indicates the tail of the list.

16. List usage

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’;
```

17. Print a list

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

Print the list head:

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

Print a node and its tail:

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

18. Recursive length computation

For the list:

```
int recursive_length() {
  if (head==nullptr)
    return 0;
  else
    return head->listlength();
};
```

For a node:

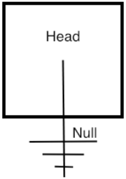
```
int listlength_recursive() {
  if (!has_next()) return 1;
  else return 1+next->listlength_recursive();
};
```

19. Iterative computation of the list length

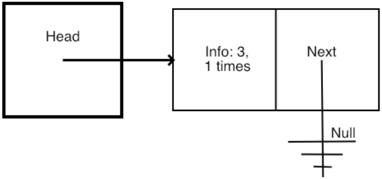
Use a shared pointer to go down the list:

```
int length_iterative() {
  int count = 0;
  auto current_node = head;
  while (current_node!=nullptr) {
    current_node = current_node->nextnode(); count += 1;
  }
  return count;
};
```

20. Creating the first list element



⇒

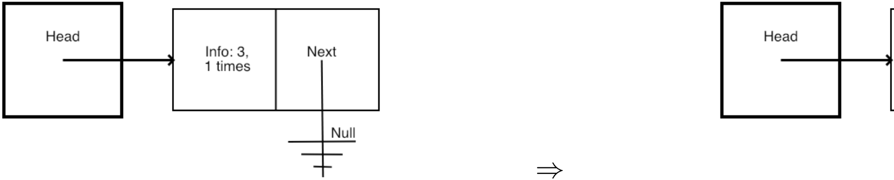


Exercise 2

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.

```
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';
```

21. Elements that are already in the list

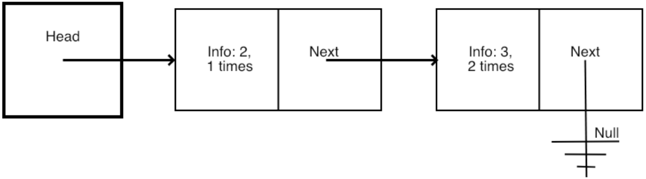


Exercise 3

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:

```
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’;
```

22. Element at the head

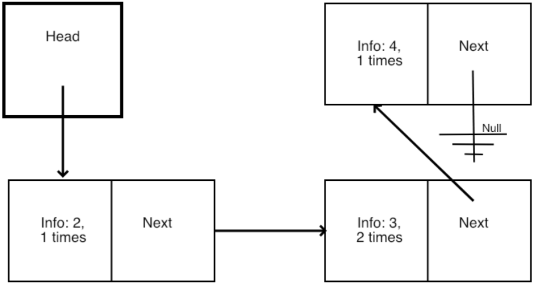


Exercise 4

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

```
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’;
```

23. Element at the tail

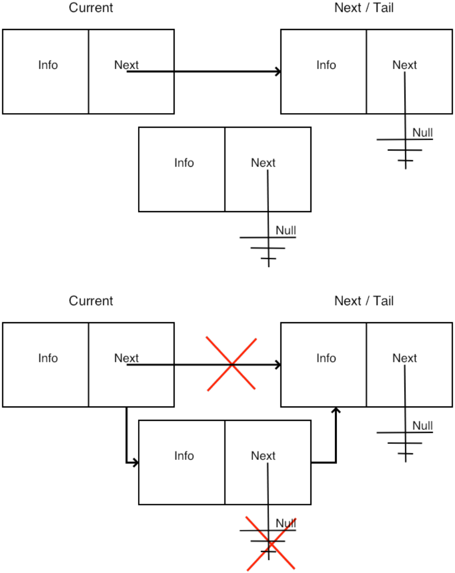


Exercise 5

If an item goes at the end of the list:

```
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’;
```

24. Insertion



Exercise 6

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

```
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’;
```