Smart Pointers

Victor Eijkhout, Susan Lindsey

Fall 2022

`last formatted: August 28, 2022`

THE UNIVERSITY OF TEXAS AT AUSTIN
Texas Advanced Computing Center

1. 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;
  /* ... */
};
```

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.

2. Simple example

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; };
};
```

3. Creating a pointer

Allocation and pointer in one:

```
shared_ptr<Obj> X =
    make_shared<Obj>( /* constructor args */ );
  // or simpler:
auto X = make_shared<Obj>( /* args */ );
```

4. Headers for smart pointers

Using smart pointers requires at the top of your file:

```
#include <memory>
using std::shared_ptr;
using std::make_shared;
// not in this lecture:
using std::unique_ptr;
using std::make_unique;
```

5. What's the point of 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
```

Automatic memory management

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.

6. Illustration

We need a class with constructor and destructor tracing:

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

7. 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
```

8. 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;
```

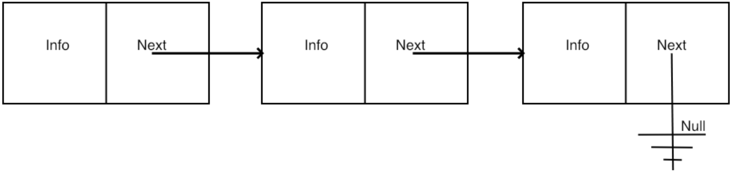
Output
[pointer] ptr2:

```
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

Linked list



9. Linked lists

The prototypical example use of pointers is in linked lists. Consider a class *Node* with

- a data value to store, and
- a pointer to another *Node*, or `nullptr` if none.

Constructor sets the data value:

```
class Node {
private:
  int datavalue{0};
  shared_ptr<Node>
    tail_ptr{nullptr};
public:
  Node() {}
  Node(int value)
    : datavalue(value) {};
  int value() { return
    datavalue; };
```

Set next / test if there is a next:

```
bool has_next() {
  return tail_ptr!=nullptr; };
```

10. List usage

Example use:

Code:

```
auto
  first = make_shared<Node>(23),
  second = make_shared<Node>(45);
first->set_tail(second);
cout << "List length: "
     << first->list_length() << "\n";
first->print();
```

Output
[tree] simple:

```
List <<23,45>> has
     length 2
```

11. Linked lists and recursion

Many operations on linked lists can be done recursively:

```
int Node::list_length() {
  if (!has_next()) return 1;
  else return 1+tail_ptr->list_length();
};
```

Exercise 1

Write a method *set_tail* that sets the tail of a node.

```
Node one;
one.set_tail( two ); // what is the type of 'two'?
cout << one.list_length() << endl; // prints 2
```

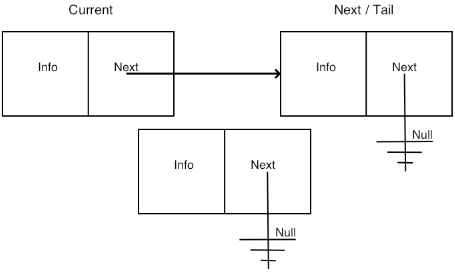
Exercise 2

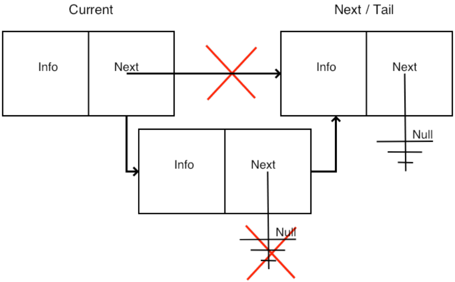
Write a recursive *append* method that appends a node to the end of a list:

```
Code:

auto
  first = make_shared<Node>(23),
  second = make_shared<Node>(45),
  third = make_shared<Node>(32);
first->append(second);
first->append(third);
first->print();
```

```
Output
[tree] append:

Append 23 & 45 gives
     <<23,45>>
Append 32 gives
     <<23,45,32>>
```

Insertion


Current
Next / Tail
Info
Next
Info
Next
Null
Info
Next
Null


Current
Next / Tail
Info
Next
Info
Next
Null
Info
Next
Null

Exercise 3

Write a recursive *insert* method that inserts a node in a list, such that the list stays sorted:

Code:

```
auto
  first = make_shared<Node>(23),
  second = make_shared<Node>(45),
  third = make_shared<Node>(32);
first->insert(second);
first->insert(third);
first->print();
```

Output
[tree] insert:

```
Insert 45 on 23
     gives <<23,45>>
Insert 32 gives
     <<23,32,45>>
```

Assume that the new node always comes somewhere after the head node.