

Smart Pointers

Victor Eijkhout, Susan Lindsey

COE 322 Fall 2020

Recursive data structures

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

This does not work: would take infinite memory.

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

Pointer 'points' to the location of the 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.

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; };  
};
```

Creating a shared pointer

Allocation and pointer in one:

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

Code:

```
HasX xobj(5);  
cout << xobj.get() << endl;  
xobj.set(6);  
cout << xobj.get() << endl;  
  
auto xptr = make_shared<HasX>(5);  
cout << xptr->get() << endl;  
xptr->set(6);  
cout << xptr->get() << endl;
```

Output

[pointer] pointx:

5
6
5
6

Headers for smart pointers

Using shared pointers requires at the top of your file:

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

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() << endl;  
yptr->set(6);  
cout << xptr->get() << endl;
```

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'.

Java/Python have 'garbage collection': runtime impact

C++ has the best solution: smart pointers.

Reference counting illustrated

We need a class with constructor and destructor tracing:

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

Pointer overwrite

Let's create a pointer and overwrite it:

Code:

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

Output

[pointer] ptr1:

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

Pointer copy

Code:

```
cout << "set pointer2" << endl;
auto thing_ptr2 =
    make_shared<thing>();
cout << "set pointer3 by copy"
    << endl;
auto thing_ptr3 = thing_ptr2;
cout << "overwrite pointer2"
    << endl;
thing_ptr2 = nullptr;
cout << "overwrite pointer3"
    << endl;
thing_ptr3 = nullptr;
```

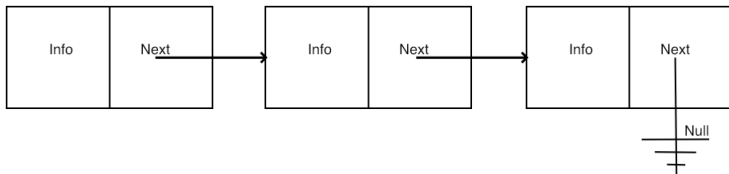
Output

[pointer] ptr2:

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

Example: linked lists

Linked list



Linked lists

The prototypical example use of pointers is in linked lists. Let a class *Node* be given:

```
class Node {  
private:  
    int datavalue{0};  
    shared_ptr<Node>  
        tail_ptr{nullptr};  
public:  
    Node() {}  
    Node(int value)  
        : datavalue(value) {};  
    bool has_next() {  
        return tail_ptr!=nullptr;  
    };  
    void set_tail  
        ( shared_ptr<Node> tail )  
        {  
            tail_ptr = tail; };  
};  
  
std::string string() {  
    stringstream strung;  
    strung << datavalue;  
    if (has_next()) {  
        strung << "," << tail_ptr->  
            string();  
    }  
    return strung.str();  
};
```

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() << endl;
first->print();
```

Output

[tree] simple:

List <<23,45>> has length 2

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 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();
```

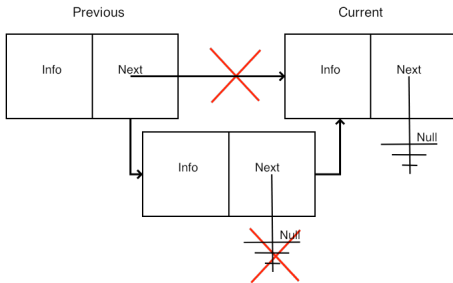
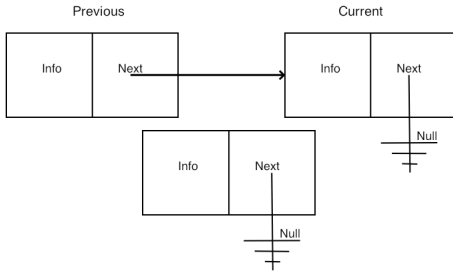
Output

[tree] append:

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

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

Insertion



Exercise 2

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>>