# C++ Part 3

CSC 230 : C and Software Tools

NC State Department of Computer Science

# Topics for Today

- Leftover topic: access restrictions
- Constructors
- Dynamic memory allocation
- Special member functions
- Overloaded operators
- Friendship
- Suppressing value semantics
- Separating the Implementation

# Keeping Secrets

- C++ offers public, protected and private access restrictions.
  - It uses a different syntax to change access
  - If we encapsulate, we're going to need public constructors and other member functions to access fields.

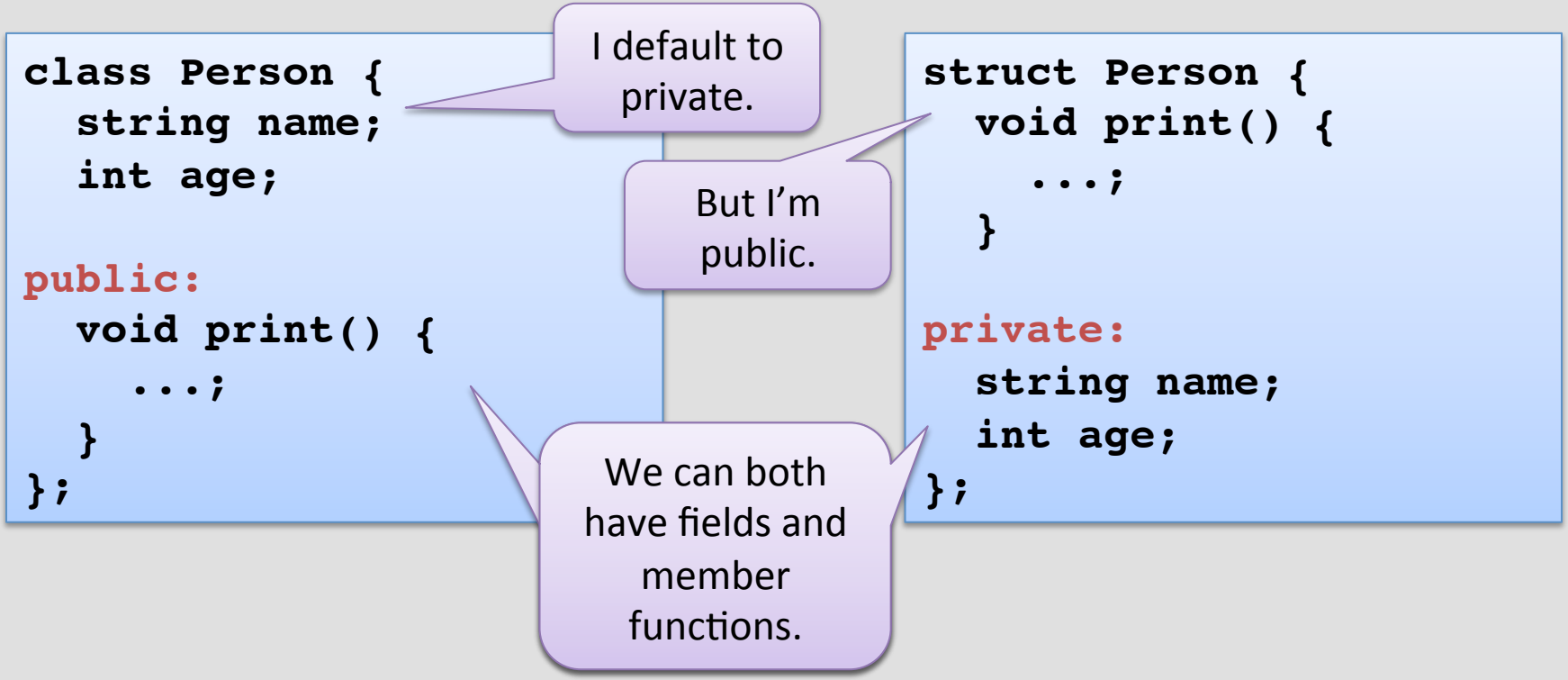
```
class Person {
private:
   string name;
   int age;

public:
   void print() {
      ...;
   }
};
```

The following members are private.

Now, some public members.

# class vs. struct

- In C++, there's only one difference between class and struct, the default access restrictions.



# class vs. struct

- But, there are conventions
  - Use class when you expect encapsulation
  - Use struct when you don't.
- Classes will have constructors and lots of member functions
- Structs may not have any (but they can)

# Constructors

- If fields are private, we need a constructor to initialize them.
  - A constructor is like a function member.
  - It has the same name as the class.
  - It has no return type.

```
class Person {
  string name;
  int age;

public:
  Person( string name, int age ) {
    this->name = name;
    Person::age = age;
  }
  ...;
};
```

The usual name collision problem. We have a this pointer, to work around it.

Or, the scope resolution operator can help.

# Calling a Constructor

- You can pass parameters to a constructor when you declare a variable:

```
class Person {
  ...;

public:
  Person( string name, int age ) {
    ...;
  }
  ...;
};
```

Make a Person variable.

```
  Person mike( "Mike", 47 );
```

```
  Person( "Tim", 7 ).print();
```

Make then use just a Person value.

We're making a value, without a variable to hold it.

# Meet new and delete

- In C, we use malloc()/free() to manage dynamically allocated memory.
- In C++, we have operators for that:
  - `new` : to allocate memory
  - `delete` : to free it
- You can use new like this to allocate space for just one value:

```
double *dPtr = new double;
```

I'm like
malloc( sizeof( double ) );

- And then delete to free that space.

```
delete dPtr;
```

I'm like
free( dPtr );

- Although, it's uncommon to dynamically allocate space for just one value.

# Allocating and Freeing Objects

- When we dynamically allocate an object, we can pass parameters to its constructor:

```
string *str = new string( "Some String" );
```

- We can use this syntax on primitive types also:

```
int *iPtr = new int( 702 );
```

- You free this memory the same way:

```
delete str;
delete iPtr;
```

# Allocating and Freeing Arrays

- There's some extra syntax when allocating arrays:

```
char *str = new char [ 200 ];
int *iList = new int [ 50 ];
string *sList = new string [ 25 ];
```

- And, when freeing them:

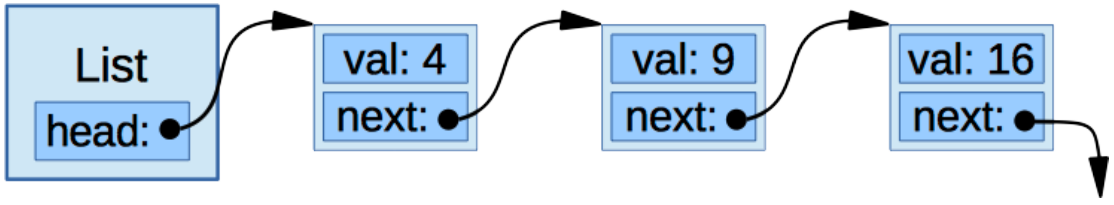
```
delete [] str;
delete [] iList;
delete [] sList;
```

# Special Member Functions

- You already know, classes can have one or more constructors
- C++ makes use of other special member functions
  - A destructor : how to reclaim resources with an instance is freed.
  - A default constructor : what to do when you make an instance without explicitly calling a constructor
  - A copy constructor : how to make an instance that's a copy of another one
  - An assignment operator : how to copy from one instance over another instance.
- Why are these special? They can be used implicitly.

# Example List Class

- To illustrate these functions, we're going to implement a list class.
  - A class containing a linked list of integers.
- This will let us show off these functions.
  - By default, we can start with an empty list.
  - To free a List, we have to free its nodes.
  - To copy a list, we need to copy all its nodes.



# List Representation and Methods

```
class List {
  struct Node {
    int val;
    Node *next;
  };

  Node *head;

public:
  ...;

  void push( int v )
  {
    Node *n = new Node;
    n->val = v;
    n->next = head;
    head = n;
  }
```

A node for building the list, scoped within the list class.

Pointer to the first node.

Add a new value at the front.

Look, the C++ version of malloc().

# List Representation and Methods

```
  int pop()
  {
    Node *n = head;
    head = head->next;

    int v = n->val;
    delete n;

    return v;
  }

  void print() {
    for ( Node *n = head; n; n = n->next )
      cout << n->val << " ";
    cout << endl;
  }
};
```

Remove a value from the front.

The C++ version of free().

Print the list to standard output.

# The Destructor

- A member function to clean up when an object is no longer needed
  - Named: ~ClassName()
  - No parameters.
- Called automatically:
  - For local variables, when they go out of scope.
  - For dynamically allocated objects, when you delete them.

```
{
    SomeClass c;
    ...;
}
```

Called here.

```
SomeClass *c = new SomeClass();
...;
delete c;
```

Called here.

# List Destructor

- For our List, destroying a list requires freeing all of its nodes:

```
class List {
  ...;

  ~List()
  {
    while ( head ) {
      Node *n = head;
      head = head->next;
      delete n;
    }
  }

  ...;
};
```

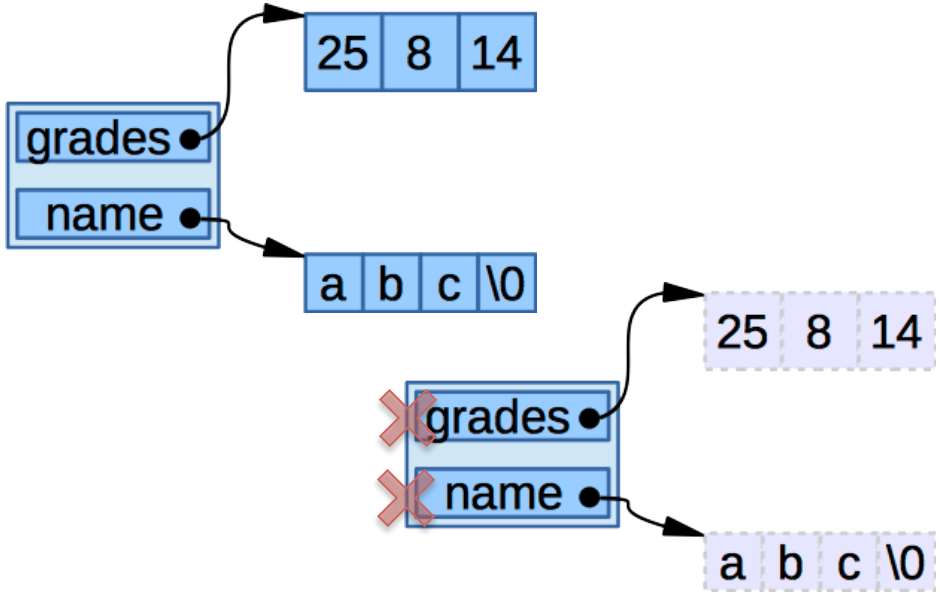
A strange name for an important function.

Free all the nodes!

# A Compiler-Generated Destructor

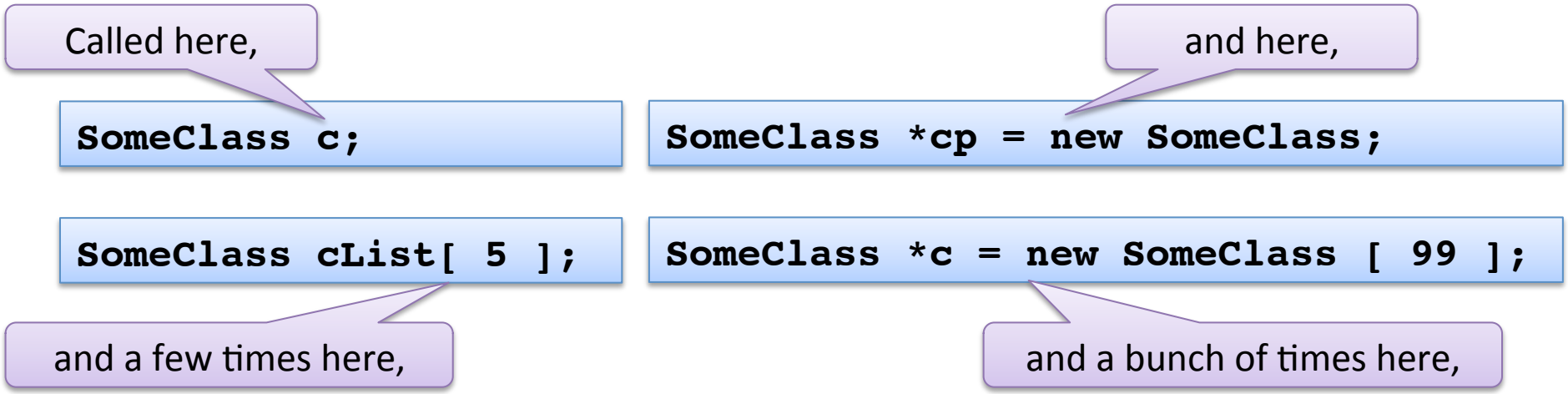
- C++ has you covered. If you don't define a destructor, the compiler will write one for you.
- What does it do?
  - It just calls the destructor on each field.
- For some classes, this is all you need.

```
class Student {
  vector< int > grades;
public:
  string name;

  ...;
}
```



# The Default Constructor

- A special constructor making an object … from nothing
  - So, it's a constructor with no parameters.
- It's called when:
  - You make an instance with no parameters.
  - You make an array of some class.



# List Default Constructor

- For our list, a default constructor can just make an empty list:

```
class List {
  ...;

  List()
  {
    head = NULL;
  }

  ...;
};
```

A constructor with no parameters.

Just set the head pointer to NULL.

Here's how we can call it.

```
  List aList;
```

# Using our List

- We can use our list like this:

```
int main()
{
  List aList;

  aList.push( 16 );
  aList.push( 9 );
  aList.push( 4 );

  aList.push( 1 );
  cout << aList.pop() << endl;

  aList.print();
  return 0;
}
```

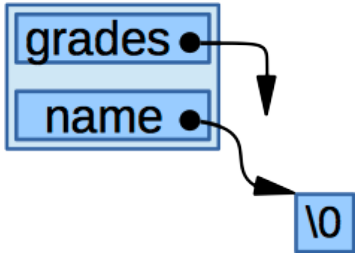
Default constructor called here.

Destructor for aList called here.

# Compiler-Generated Default Constructor

- If you don't define a default constructor, the compiler will write one for you.
- What does it do?
  - It just calls the default constructor for each field.
- For some classes, this is enough.

```
class Student {
  vector< int > grades;
public:
  string name;

  ...;
}
```



# The Copy Constructor

- A special constructor for making an object from another object of the same type.
- It's called when:
  - You make an object initialized with another object.
  - You pass another object of the same type to the constructor.
  - You pass an object (by value) to a function.

```
SomeClass a;
...;
SomeClass b = a;
SomeClass c( a );
```

Called for both of these.

```
void f( SomeClass x )
{
    ...;
}
```

```
SomeClass a;
...;
f( a );
```

Also called here.

# List Copy Constructor

```
class List {
  ...;

  List( const List &other )
  {
    Node **ptr = &head;

    for ( Node *n = other.head; n; n = n->next ) {
      *ptr = new Node;
      (*ptr)->val = n->val;
      ptr = &((*ptr)->next);
    }

    *ptr = NULL;
  }

  ...;
};
```

A copy constructor will always have parameters like this.

Notice, pass-by-reference here. Important.

Copy all the nodes on the list.

This will call it (although passing a list to a function is more common).

```
class bList = aList;
```

# A Bad Copy Constructor

```
class List {
  ...;

  List( List other )
  {
    Node **ptr = &head;

    for ( Node *n = other.head; n; n = n->next ) {
      *ptr = new Node;
      (*ptr)->val = n->val;
      ptr = &((*ptr)->next);
    }

    *ptr = NULL;
  }

  ...;
};
```

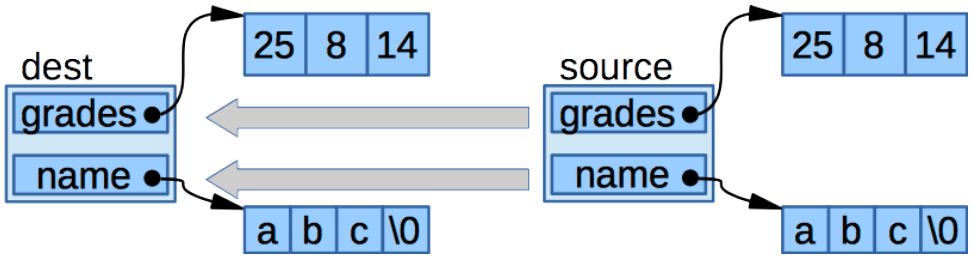
There's your problem, right there.

This function needs to call the copy constructor just to pass this parameter.

# Compiler-Generated Copy Constructor

- If you don't define a copy constructor, the compiler will write one for you.
- What does it do?
  - It just calls the copy constructor for each field.
- For some classes, this is enough.

```
class Student {
  vector< int > grades;
public:
  string name;

  ...;
}
```



# Overloading Operators

- C++ lets us define what the standard operators should do, when used with non-standard types.
  - We still have the same set of operators, with the same precedence and associativity
  - We can just change what they do in particular contexts.
- We define functions to be called when various operators are used.

```
C operator+( A a, B b )
{
   ...;
}
```

They can be a free function.

```
class A {
   ...;

   C operator+( B b ) {
      ...;
   }
}
```

Or a member function, where the object is the left operand.

# List Overloaded [ ]

```
class List {
  ...;

  int &operator[]( int i )
  {
    Node *n = head;
    while ( i > 0 ) {
      n = n->next;
      i--;
    }

    return n->val;
  }

  ...;
};
```

Return a reference to the value in the $i^{th}$ node.

This type isn't required to be an int; you can index by other types if you want.

Skip past nodes until we get to the $i^{th}$ one.

Return a reference to the val field in this node.

```
  int x = aList[ 1 ];
  bList[ i ] = 25;
```

You can use this to get or set the values in the list.

# List Overloaded <<

- We can overload <<, to print our class just like a primitive type.

```
class List {
  ...;
};

ostream &operator<<( ostream &output, const List &lst ) {
  for ( List::Node *n = lst.head; n; n = n->next )
    output << n->val << " ";

  return output;
}
```

This has to be a free operator function, it can't be a member of list.

Why?

```
  cout << aList << endl;
```

We'll be able to call this operator like this.

# List Overloaded <<

- This code needs some explaining.

```
class List {
  ...;
};

ostream &operator<<( ostream &output, const List &lst ) {
  for ( List::Node *n = lst.head; n; n = n->next )
    output << n->val << " ";

  return output;
}
```

This is the type for output streams (like cout)

So, when << is used with an output stream on the left and a List on the right, use this function.

We return a copy of the output stream, so you can stack up multiple uses of <<.

```
  cout << aList << endl;
```

This is evaluated as:
( cout << aList ) << endl;

# List Overloaded <<

- This code needs more explaining.

```
class List {
  ...;
};

ostream &operator<<( ostream &output, const List &lst ) {
  for ( List::Node *n = lst.head; n; n = n->next )
    output << n->val << " ";

  return output;
}
```

We're outside the class, so we must use scope resolution to talk about the Node in List.

But, if we're outside List, can we access this private field?

```
  cout << aList << endl;
```

No. No we can't.

# Friendship

- We can selectively give away access to our private parts.
  - We can declare an outside function or class as a friend.

```
class List {
  friend ostream &operator<<( ostream &output, const List &lst );

  ...;
};

ostream &operator<<( ostream &output, const List &lst ) {
  for ( List::Node *n = lst.head; n; n = n->next )
    output << n->val << " ";

  return output;
}
```

"This function is part of my implementation."

Now this is OK.

# The Assignment Operator

- A special function for copying one object over an existing object.
- It's called when:
  - You assign to an object.
- How is this different from the copy constructor?
  - Here, you already have a destination object; you just need to replace it.

```
SomeClass a;
SomeClass b;
...;
a = b;
```

Assignment operator.

```
SomeClass a;
SomeClass b = a;
```

Copy constructor.

# List Assignment Operator

```
class List {
  ...;
  List &operator=( const List &other )
  {
    if ( this != &other ) {
      while ( head ) {
        Node *n = head;
        head = head->next;
        delete n;
      }

      Node **ptr = &head;
      for ( Node *n = other.head; n; n = n->next ) {
        *ptr = new Node;
        (*ptr)->val = n->val;
        ptr = &((*ptr)->next);
      }
      *ptr = NULL;
    }

    return *this;
  }
  ...;
};
```

Protection in case we say: a = a;

Free the old list.

Copy over the source list.

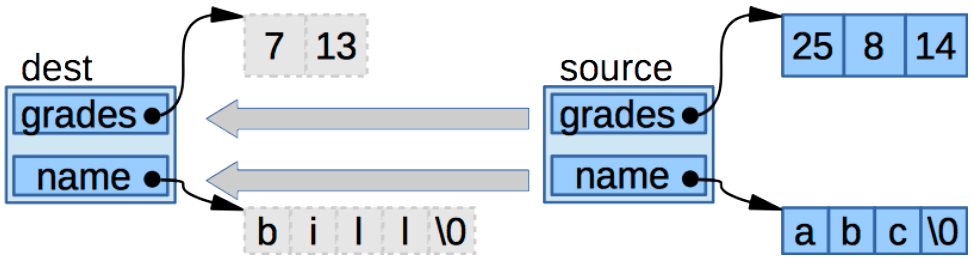
Return the resulting list by reference.
In case we want to say: a = b = c;

Looks a lot like a destructor then a copy constructor.

# Compiler-Generated Assignment Operator

- If you don't define an assignment operator, the compiler will write one for you.
- What does it do?
  - It just calls the assignment operator for each field.
- For some classes, this is enough.

```
class Student {
  vector< int > grades;
public:
  string name;

  ...;
}
```



# List Overloaded +

- I also overloaded + so it concatenates lists.

```
class List {
  friend List operator+( const List &a, const List &b );

  ...;
};

List operator+( const List &a, const List &b )
{
  ...;
}

List cList = aList + bList;
```

Need friendship to access List representation.

And here's the operator definition.

We can use it like this.

- Really, nothing new here, but I need it for the next topic.

# Constructors and Type Conversion

- C++ has a special interpretation for one-parameter constructors.
  - We can use them to implicitly convert types.

```
class SomeClass {
  ...;
  SomeClass( int a ) {
    ...;
  }

  SomeClass( const string &a ) {
    ...;
  }
  ...;
}
```

How to build this object from an int.

How to build this object from a string.

# List from Int

- Pretend we want to be able to make a List from an int.

```
class List {
  ...;
  List( int v )
  {
    head = new Node;
    head->val = v;
    head->next = NULL;
  }
  ...;
};
```

Here are the instructions.

No surprise, we can use this new constructor like this.

```
List bList = 25;
List cList = 16 + bList + 36 + 49;
```

Surprise! The compiler will also use it to be able to evaluate this expression.

# Suppressing Value Semantics

- These special members help support value semantics, the ability to pass, return and assign objects by value.
- What if we don't want that? We didn't seem to need it in Java.
  - C++ has a syntax to suppress generation of these functions.

```
class List {
  ...;
  List( const List &other ) = delete;

  List &operator=( const List &other ) = delete;
  ...;
};
```

I don't want a copy constructor.

I don't want an assignment operator.

```
  List aList, bList;
  ...;
  List cList = aList;
  aList = bList;
```

Now, we can't do this.

# Separating Implementation

- We need to be able to describe a class in a header file
- … while leaving the details to an implementation file.
- In the header, we can define the class, including:
  - All its fields
  - Prototypes for its methods

```
class List {
  struct Node {
    int val;
    Node *next;
  };
  Node *head;
public:
  List();
  List( const List & );
  ~List();
  ...;
}
```

Here's my representation.

I have these functions, but they're defined elsewhere.

# Separating Implementation

- In the implementation file, we just need to define all the methods.

```
#include "List.h"
#include <iostream>

using namespace std;

List::List()
{
  head = NULL;
}

List::List( const List &other )
{
  Node **ptr = &head;

  ...;
```

Include my own header file first, a good policy.

I'm defining the default constructor that's part of the List class.

I'm defining the copy constructor that's part of the List class.