Data Structures and Object Orientation

CSC 230 : C and Software Tools

NC State Department of Computer Science

Topics for Today

- Leftover topic: The preprocessor conditional compilation
- Pointer-to-pointer, in pictures
- Tree-shaped data structures
- DIY inheritance
- DIY member functions
- DIY function overloading
- Virtual object destruction

Conditional Compilation

- To control what source code gets compiled
- Common uses
  - to resolve, at compile time, platform (machine- or OS-) dependencies
  - to compile (or not) debugging code
- Here's what conditional compilation looks like

```
#if...

someCodeYouMay();
orMayNotWant();

#endif
```

Conditional Compilation

- We have lots of (related) directives for writing condition compilation
  - **`#if / #ifdef / #ifndef`**
  - **`#elif / #else`**
  - **`#endif`**

Preprocessor Expressions

- We can ask the preprocessor if macros are defined.

```
#if defined(X)
```

- This is so common, it has its own syntax.

```
#ifdef X
```

- We can ask about particular values.

```
#if X == 25
```

- We can even build compound conditionals

```
#if X > 25 && Y < 30
```

Conditional Compilation: Example

```
#if defined(LINUX)
  #define HDR "linux.h"
#elif defined(WIN32)
  #define HDR "windows.h"
#else
  #define HDR "default.h"
#endif

#include HDR
```

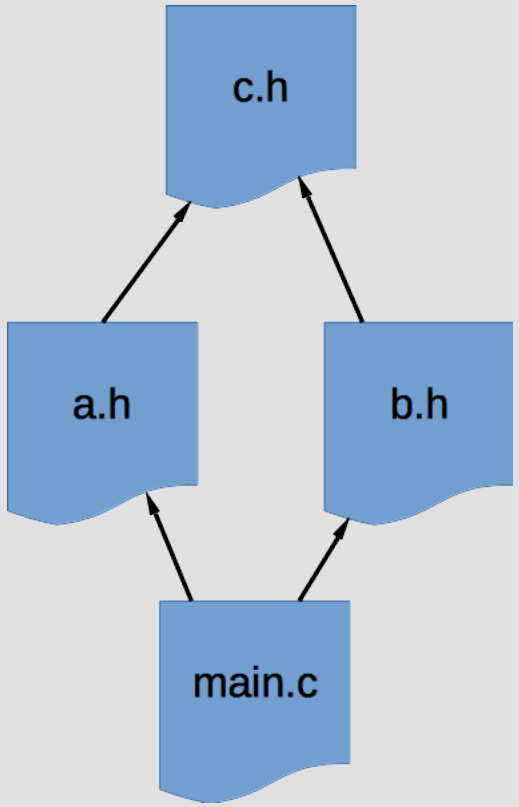
- And when compiling this program, can define what **SYSTEM** is by using the **-D** option to **gcc**

```
gcc -DWIN32 myprog.c …
```

```
#include "windows.h"
```

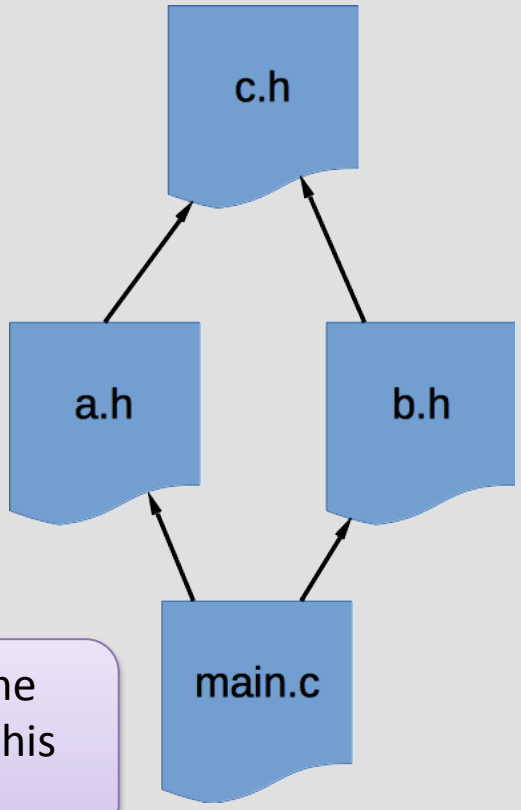
Include Guards

- It's possible to include the same header more than once … even without trying to.
- Is this bad?
  - Well, that's extra work for the compiler
  - And, if the header contains definitions, we're in trouble.
  - You can declare something as many times as you like, but you better only define it once.
- How can we fix this?
  - You guessed it, with the preprocessor.



Include Guards

- We can use trick the preprocessor into discarding the contents of a header after it's already been processed once.




```
#ifndef C_H
#define C_H

…
… all the stuff inside c.h
…

#endif
```

Some name unique to this header.

Help with Debugging

- The preprocessor can help with debugging
- We can use it to exclude blocks from compilation.

```
#ifdef DEBUG

   …;
   #ifdef DEBUG
      …;
   #endif
   …;

#endif
```

Unlike comments, these will nest.

Help with Debugging

- Here, call-by-name lets us do things we couldn't with a function.
- We can give more context in our debug output
- Or, disable them completely with a recompile.

```
#if defined(DEBUG)
#define REPORT_I( X ) printf( "%s = %d\n", #X, (X) )
#else
#define REPORT_I( X )
#endif

…

  REPORT_I( i );
  REPORT_I( total );
```

Help with Debugging

- When we compile, we can choose which macros to enable.

```
gcc -DDEBUG -std=c99 -Wall …
```

Any macro you want to define.

I'm defined to **1**

```
gcc -DNAME=BILL "-DMESSAGE=HELLO WORLD" -std=c99 …
```

I'm set to BILL

With help from the shell, you can include spaces.

Linked List Representation

- Same representation as before.

```
struct NodeTag {
  int value;
  struct NodeTag *next;
};

typedef struct NodeTag Node;
```

Representation for a Node.

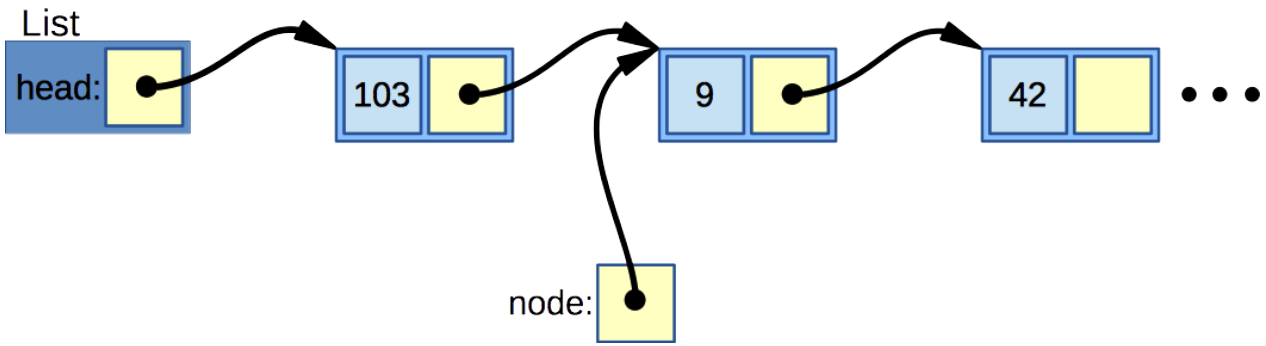
```
typedef struct {
  Node *head;

} List;
```

Representation for a whole list.

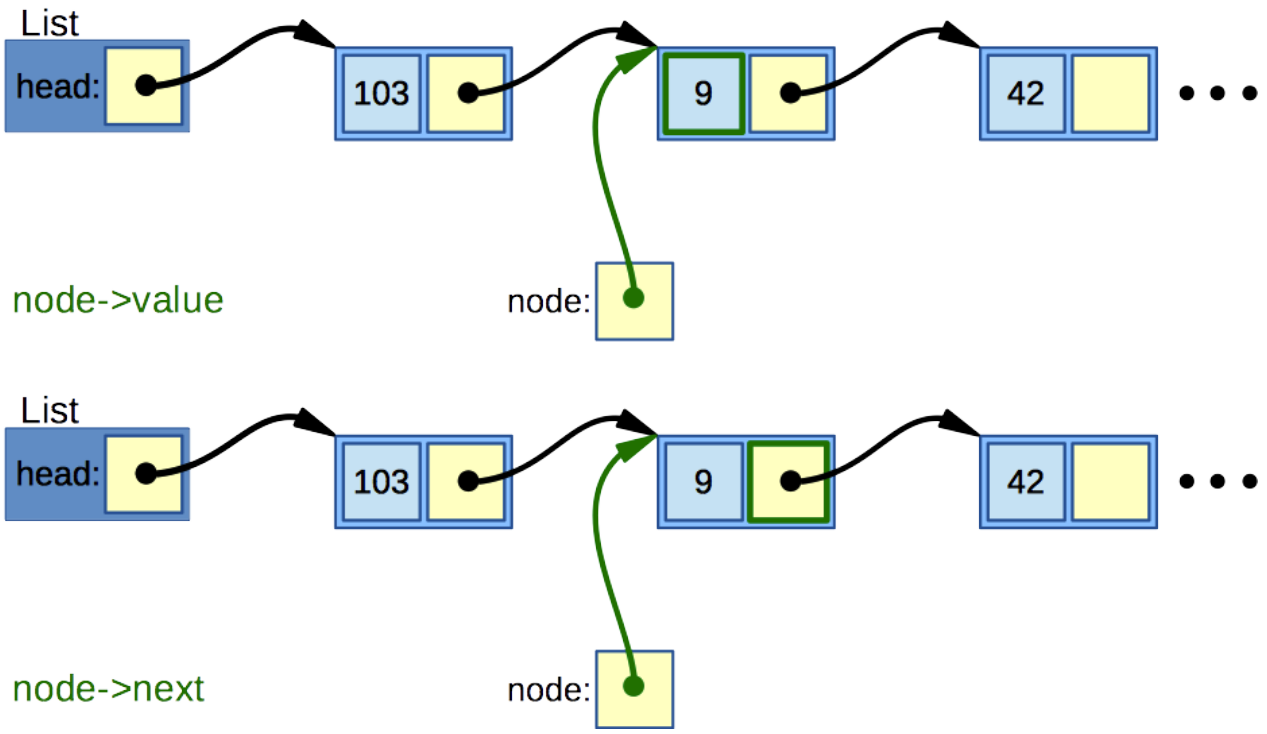
Pointer Syntax Review

- A pointer to a node:
  Node *node;
- It's good for … well … keeping up with a node.



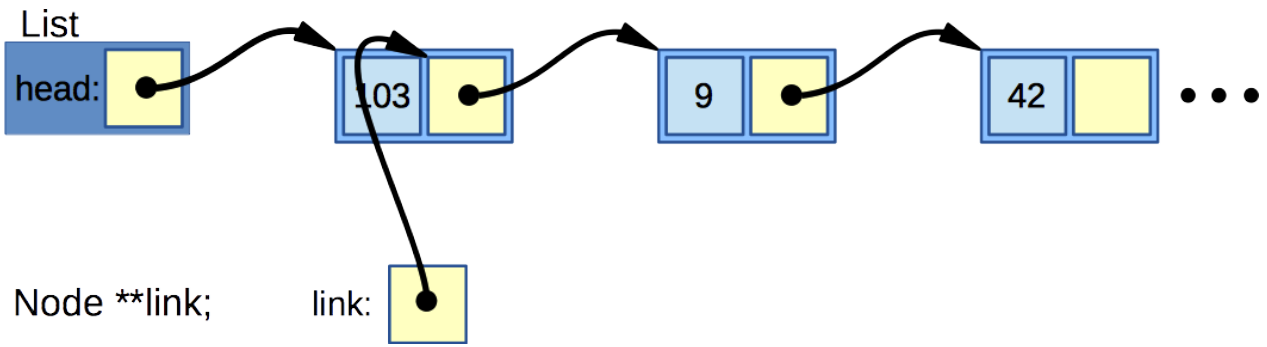
Pointer Syntax Review

- You can easily get to the fields of that node



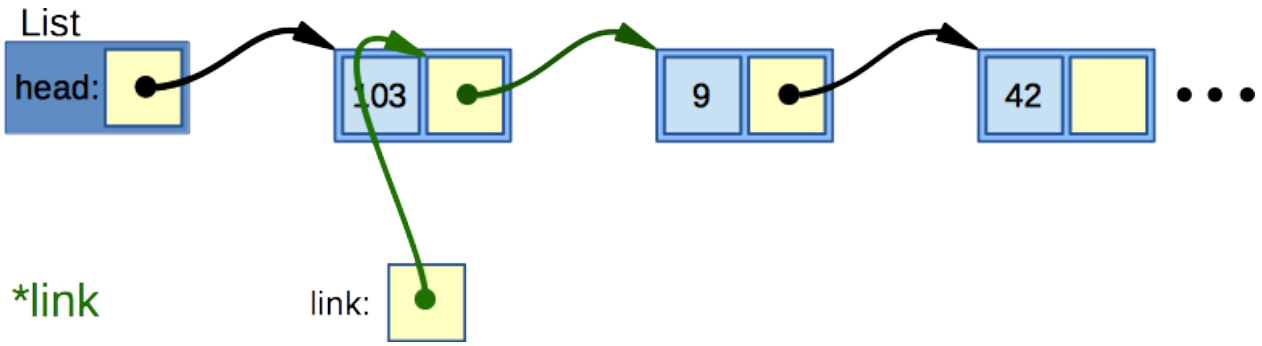
Pointer Syntax Review

- A pointer to a node won't let you remove the node itself.
- But, a pointer to a pointer, that's something more useful:



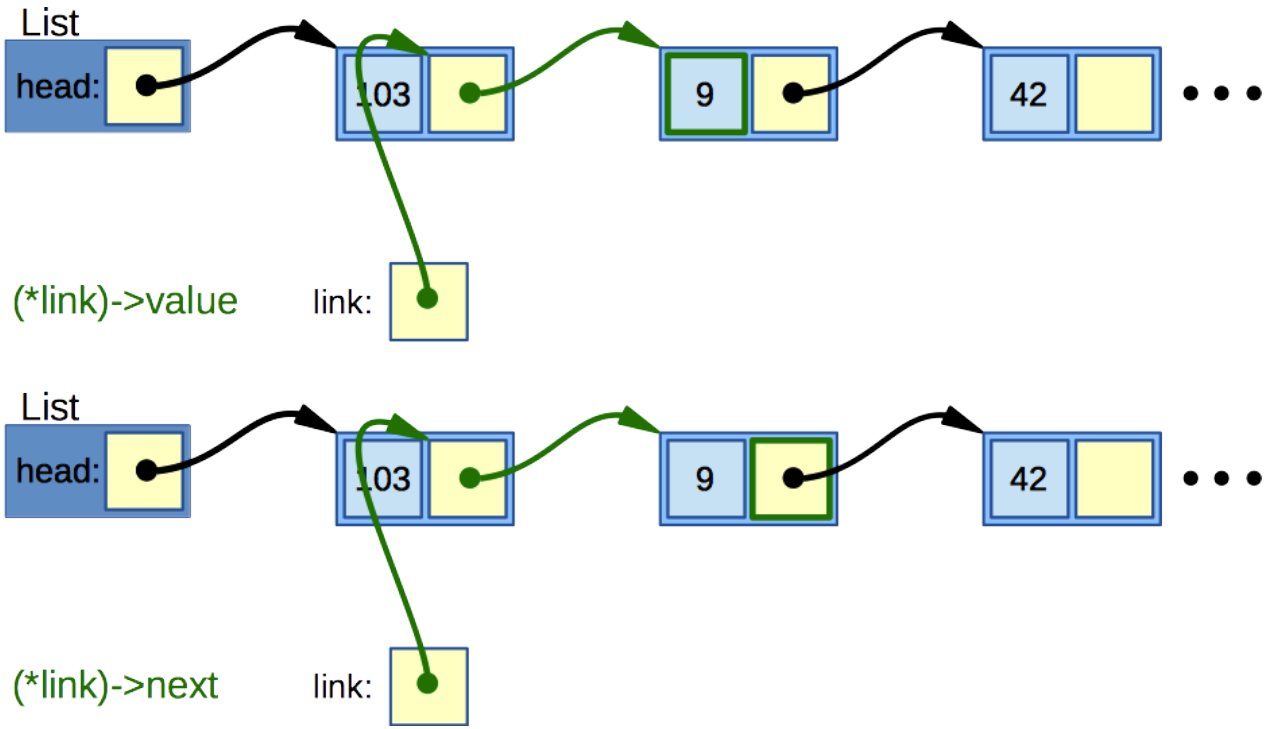
Pointer Syntax Review

- With a dereference, we can use it just like a node pointer.



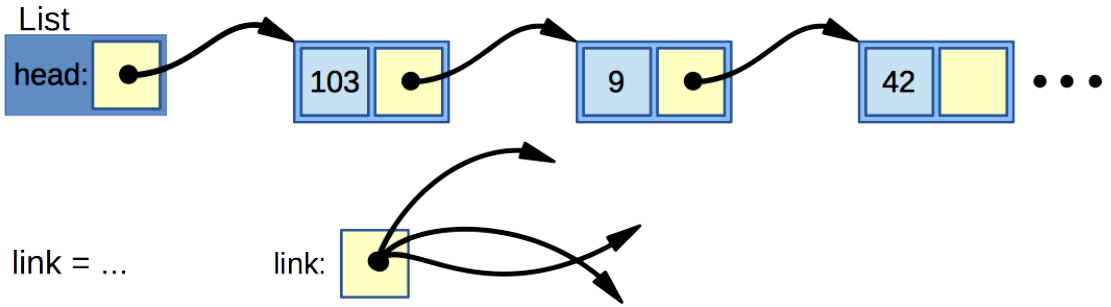
Pointer Syntax Review

- We can use this to access fields of the node:

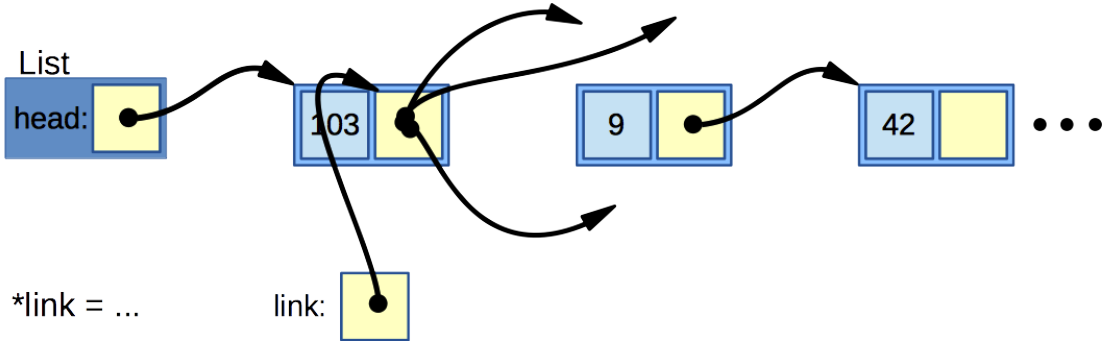


Pointer Syntax Review

- Assigning to the pointer can make it point to another link

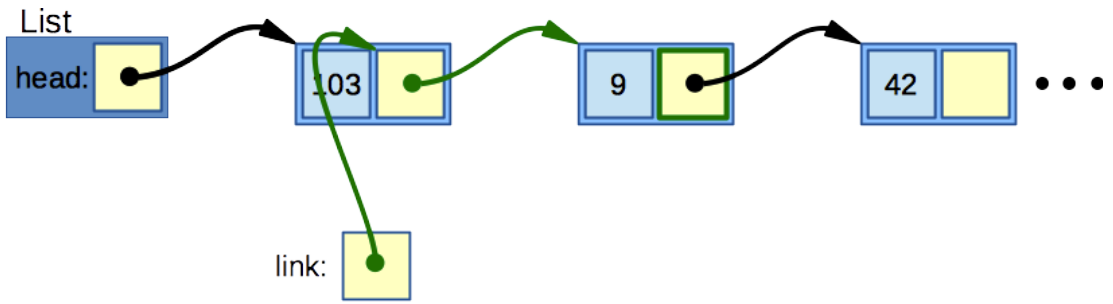


- Assigning to what it points to can let you change the linking structure.

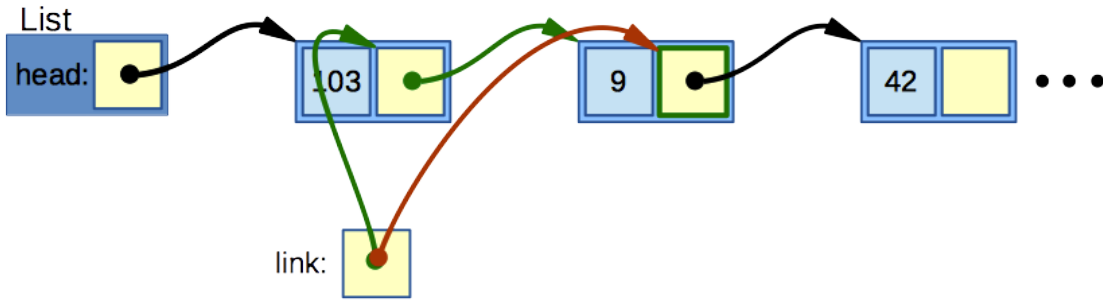


Pointer Syntax Review

- We can move our link pointer ahead by one node:



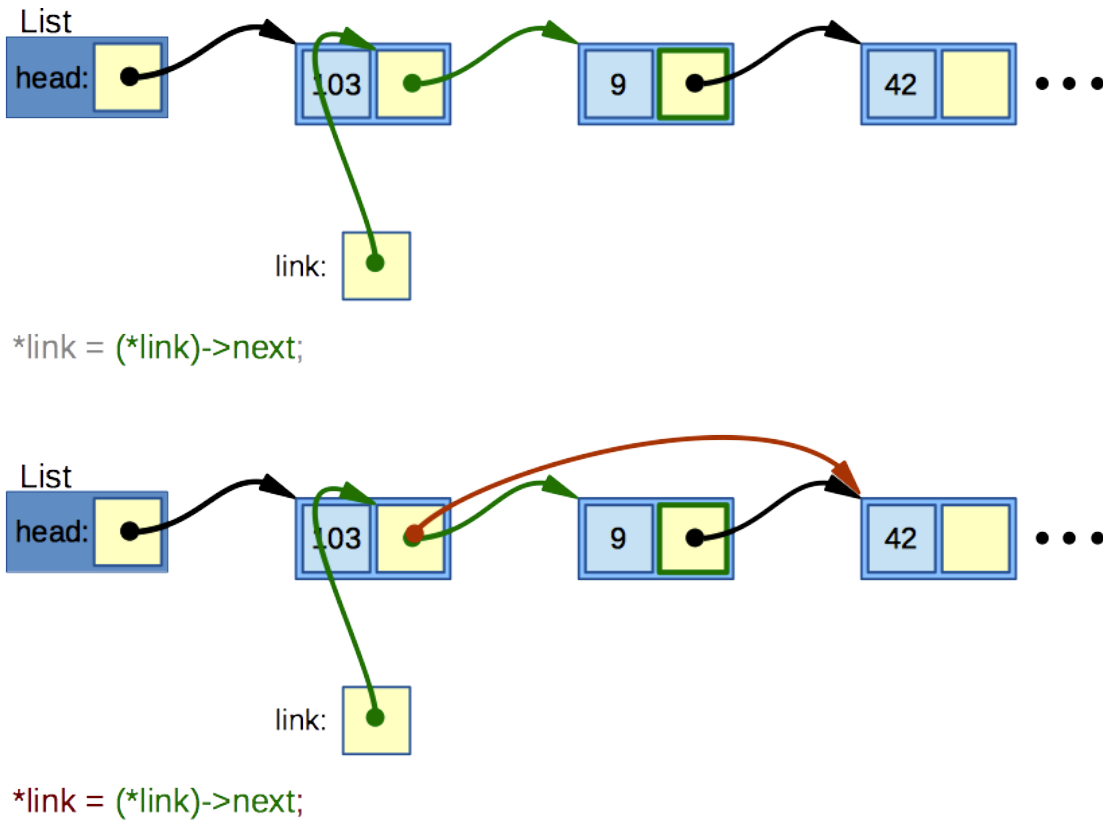
link = &((*link)->next);



link = &((*link)->next);

Pointer Syntax Review

- With dereference, we can change the linking structure:



Here, we're removing a node, but we could easily insert one instead.

Building a Tree

- This technique can simplify code for many linked structures.
- A binary tree? No problem.
- Here's a representation:

```
struct NodeTag {
  int value;
  struct NodeTag *left, *right;
};

typedef struct NodeTag Node;

typedef struct {
  Node *root;
} Tree;
```

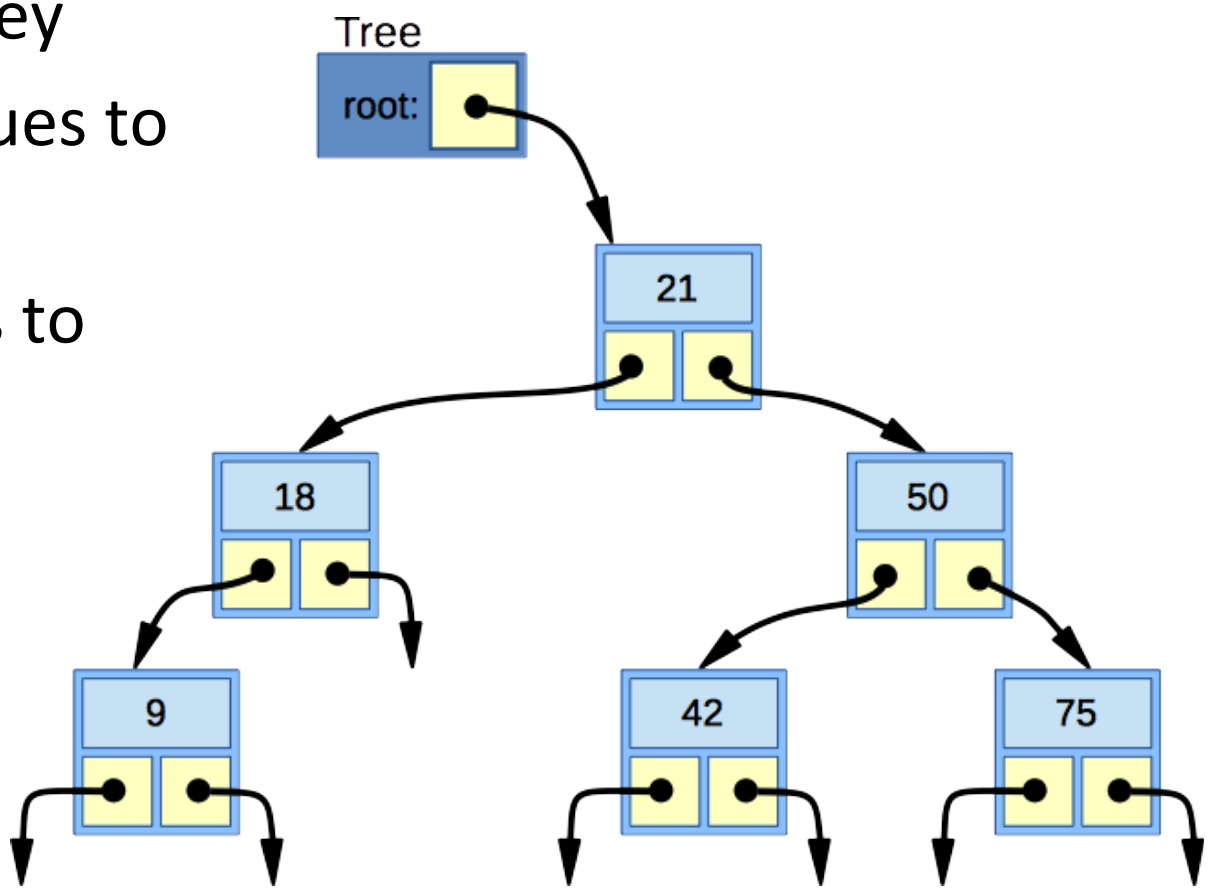
Structure for a node.

Short name for that structure

Object containing the tree.

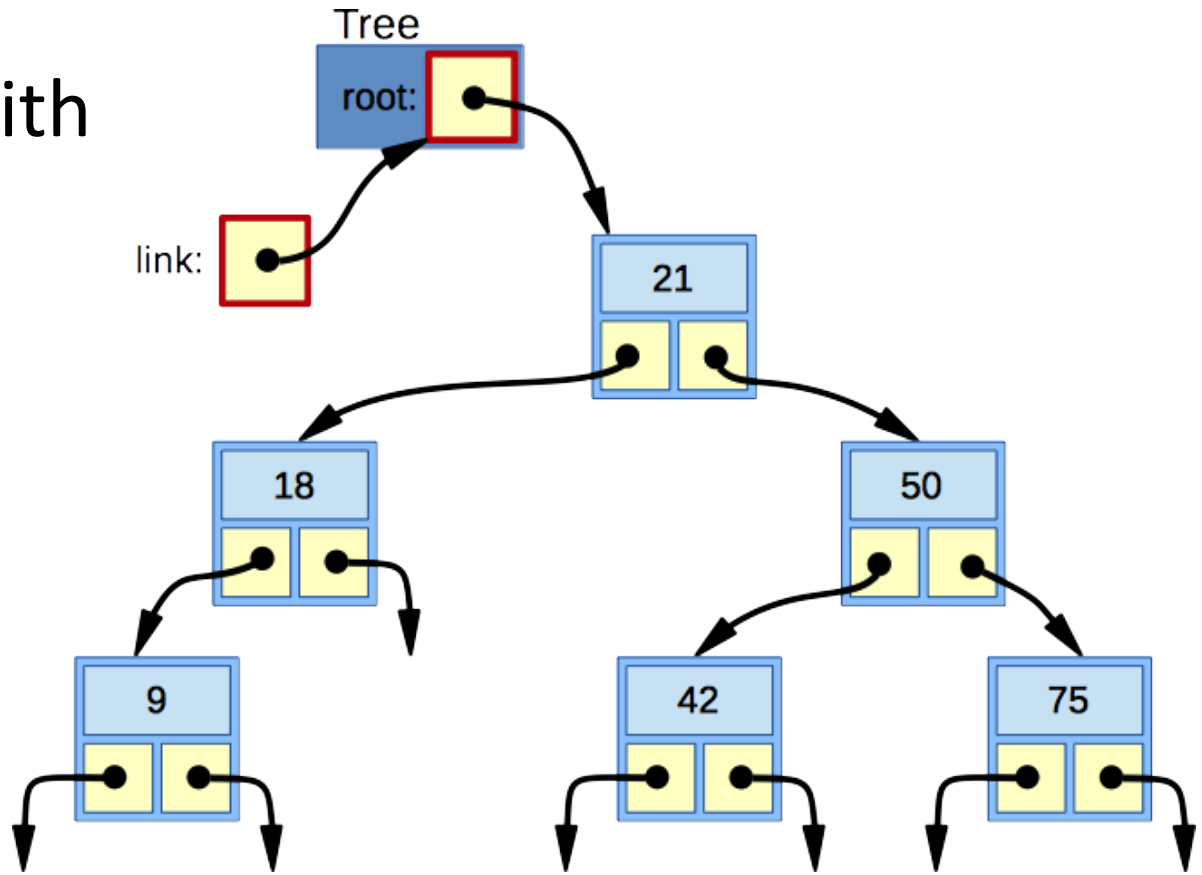
Tree Structure

- It's your standard binary search tree:
  - Sorted by key
  - Smaller values to the left
  - Larger ones to the right.



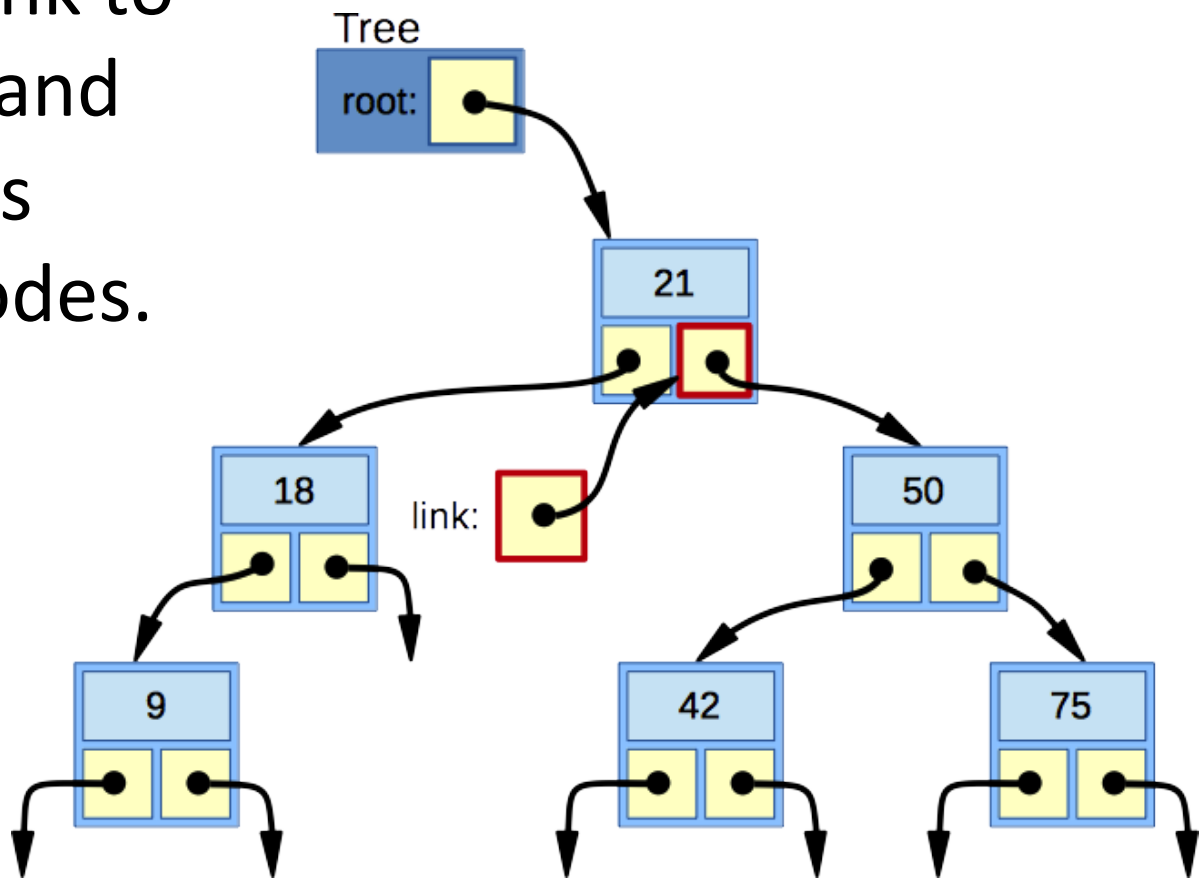
Inserting Values

- On insert, we'll use pointer to pointer to Node.
- We'll start with a pointer to the root pointer
- Say we're inserting 30



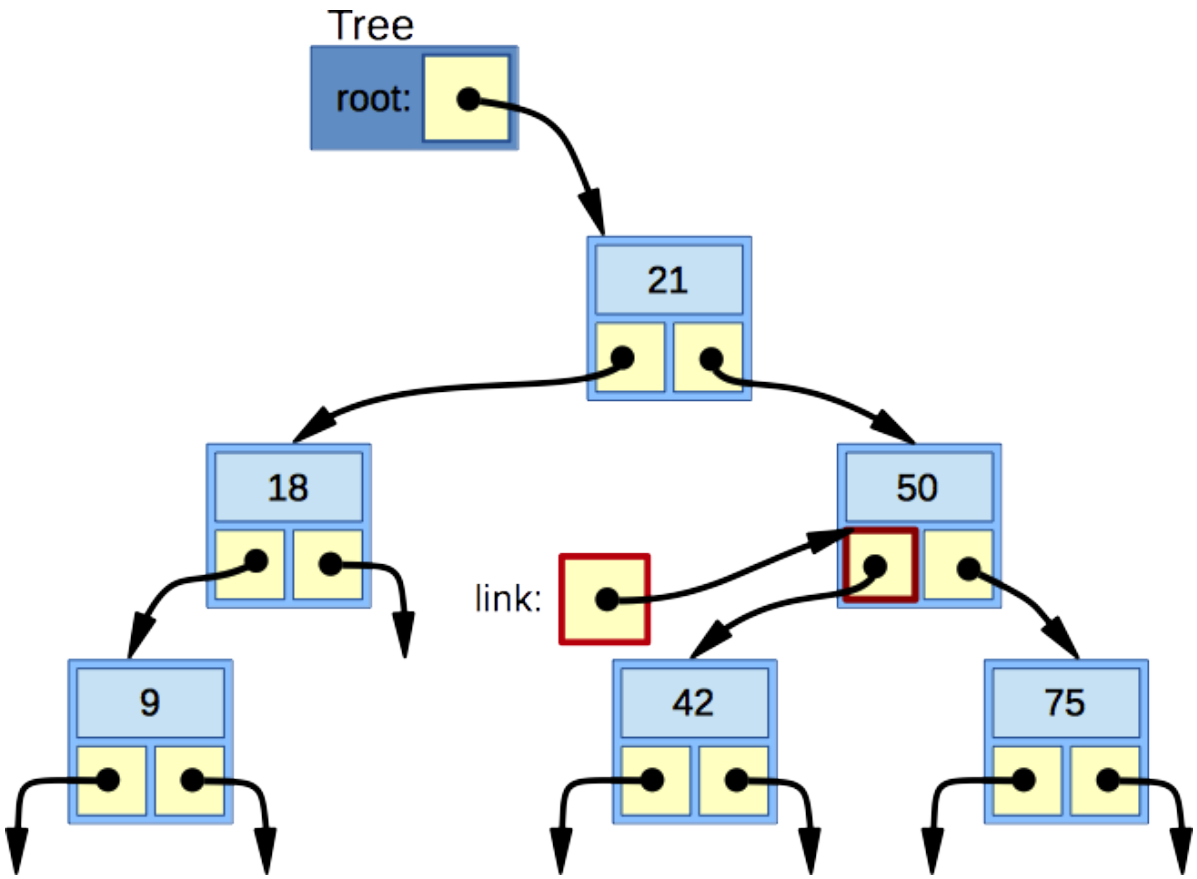
Inserting Values

- As we look for where to put the new value, we'll move link to point to left and right pointers inside the nodes.



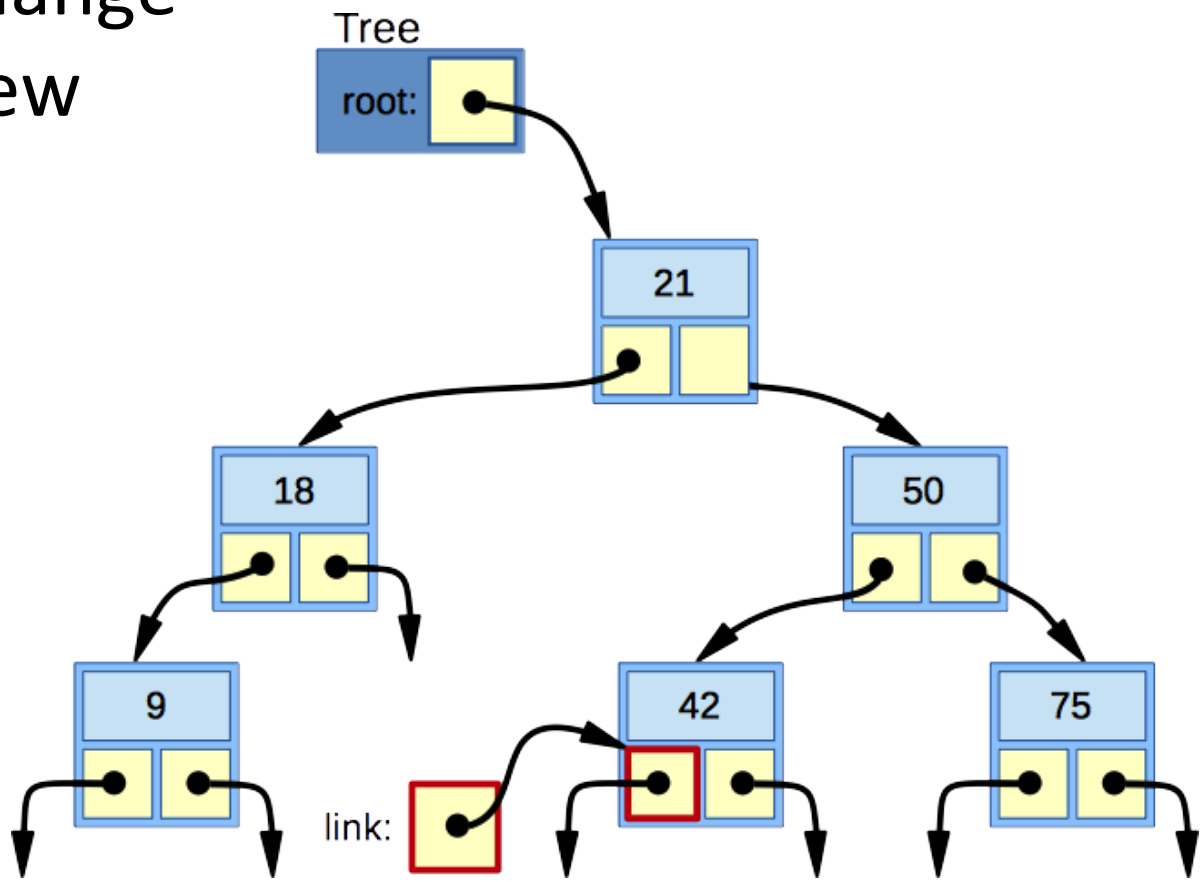
Inserting Values

- And so on.



Inserting Values

- When we reach a pointer to null, that's the pointer to change to insert a new node.



Tree Insertion, Simplified

```
bool insert( Tree *tree, int val )
{
  Node **link = &tree->root;

  while ( *link ) {
    if ( (*link)->value == val )
      return false;
    if ( val < (*link)->value )
      link = &(*link)->left;
    else
      link = &(*link)->right;
  }

  Node *n = (Node *)malloc( sizeof( Node ) );
  n->value = val;
  n->left = n->right = NULL;

  *link= n;
  return true;
}
```

Start pointing to the root pointer.

Descend the tree, pointing to the left/right pointers in the nodes.

Allocate a new leaf for this value.

Link it in as a child of some parent.

Object Orientation in C

- You can write object-oriented code in C
  - C programmers used to do this all the time before C++ was created.
- You just have to do a lot of the work yourself
  - This is a little bit painful
  - But worthwhile … it exposes how techniques like inheritance and overriding really work.

An Example of Object-Orientation

- We're going to build classes to represent cells in a spreadsheet.

```
+----------+------------+-----------------+
|          |       3.75|              Gum|
+----------+------------+-----------------+
|          |       8.90|         Sandwich|
+----------+------------+-----------------+
```

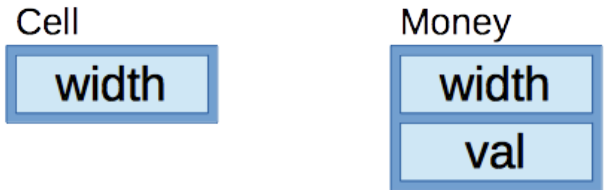
A two-dimensional array of Cell objects.

Different types of cells will store different values.

And print themselves differently.

Inheritance Exposed

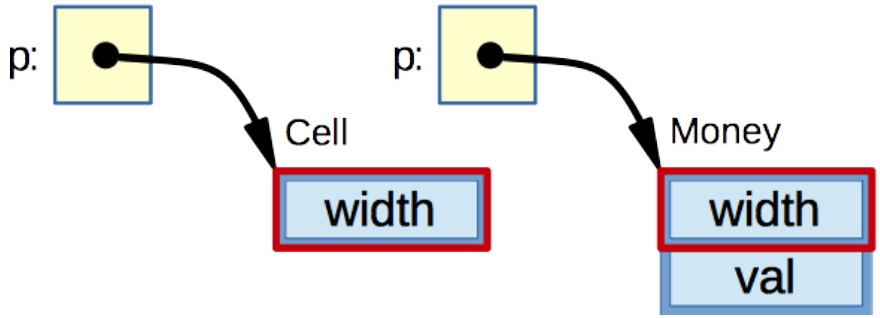
- We'll start with just two types of cells
  - Cell : a superclass that has a width field, but doesn't display anything.
  - Money : a subclass of Cell that also maintains and displays a monetary amount.
- We'll build these so they look like this in memory:



- Money starts out just like Cell
  - But it adds a field later in memory.

Inheritance Exposed

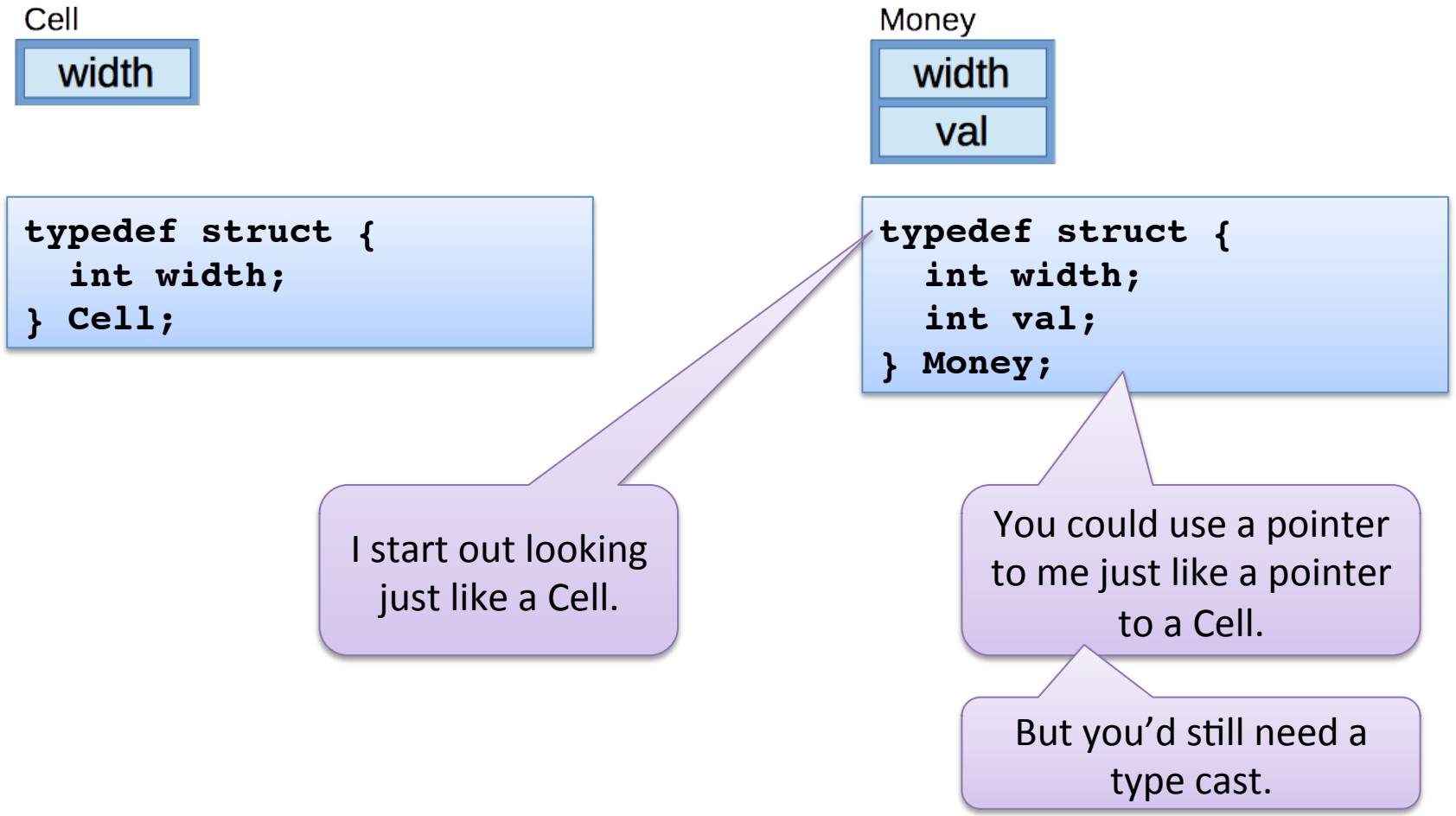
- So, a pointer to a Money instance looks just like a Cell instance.
  - It has all the fields you'd expect in a Cell.
  - In exactly the same layout



- This is how inheritance works.
  - Fields of the superclass are placed at the start of the subclass' fields.
  - With exactly the same layout in memory.

Inheritance In C

- Here's how we could make this work in C

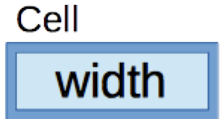


Constructors in C

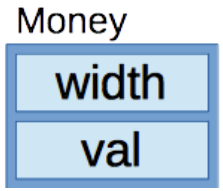
- For a constructor, we just need a function that can:
  - Create each type of object.
  - And return a pointer to it.

```
Cell *makeCell( int width )
{
  Cell *this = (Cell *)malloc( sizeof( Cell ) );
  this->width = width;
  return this;
}
```

Not a reserved word in C, but it helps show what we're doing.




```
Money *makeMoney( int width, int val )
{
  Money *this = (Money *)malloc( sizeof( Money ) );
  this->width = width;
  this->val = val;
  return this;
}
```



Using Objects

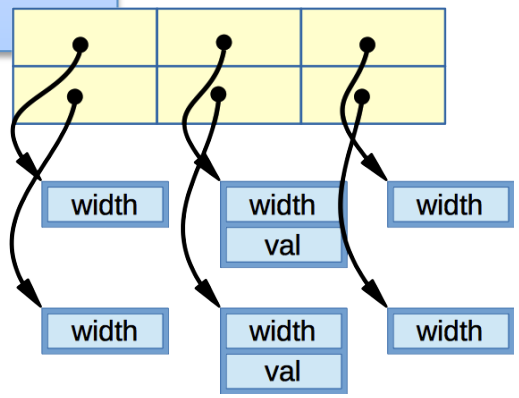
- In client code, we can use a Cell pointer to hold the address of either kind of object.

```
Cell *sheet[ 2 ][ 3 ];

sheet[ 0 ][ 0 ] = makeCell( 10 );
sheet[ 0 ][ 1 ] = (Cell *) makeMoney( 12, 375 );
sheet[ 0 ][ 2 ] = makeCell( 16 );

sheet[ 1 ][ 0 ] = makeCell( 10 );
sheet[ 1 ][ 1 ] = (Cell *) makeMoney( 12, 890 );
sheet[ 1 ][ 2 ] = makeCell( 16 );
```

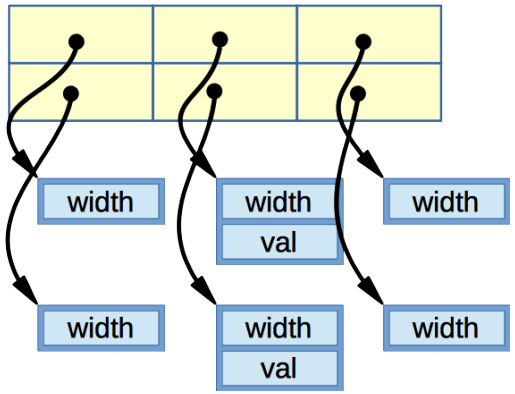
The compiler doesn't know that a Money pointer can be used as a cell pointer.



Using Objects

- In client code, we can expect the same thing from both types of objects, a width field.

```
for ( int i = 0; i < 2; i++ ) {
  for ( int j = 0; j < 3; j++ ) {
    Cell  *c = sheet[ i ][ j ];
    printf( "|%*s", c->width, "" );
  }
  printf( "|\n" );
}
```



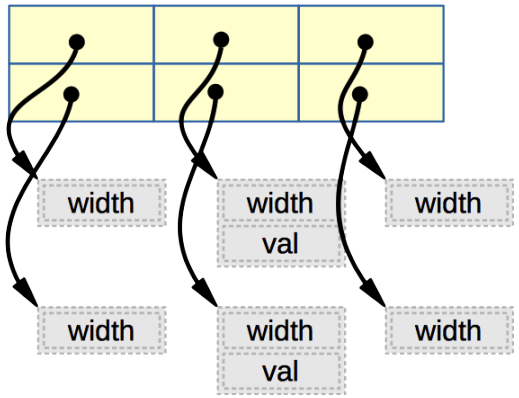

```
+----------+-----------+----------------+
|          |           |                |
+----------+-----------+----------------+
|          |           |                |
+----------+-----------+----------------+
```

With some additional decoration, we get this output.

Cleaning Up

- When we're done with the objects, we can release their memory with free( ).

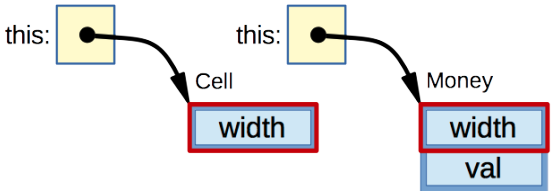
```
for ( int i = 0; i < 2; i++ ) {
  for ( int j = 0; j < 3; j++ ) {
    free( sheet[ i ][ j ] );
  }
}
```



Methods

- A method : function that operates on an instance of an object.
- We can do that. We just need to pass in the instance the function should operate on.
  - Internally, C++ and Java do this also.

```
int getWidth( Cell *this )
{
  return this->width;
}
```

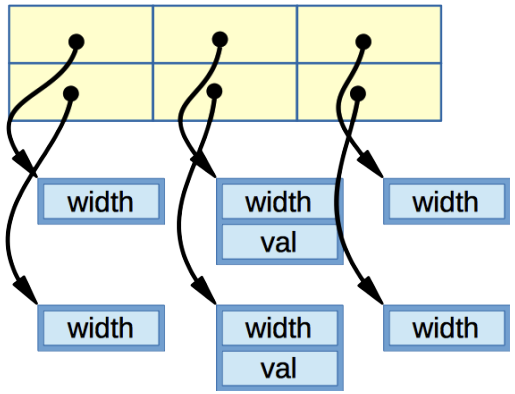


- This function will work for instance of Cell or Money (they both have a width field)

Calling Methods

- In client code, we can call this method instead of accessing fields directly.

```
for ( int i = 0; i < 2; i++ ) {
   for ( int j = 0; j < 3; j++ ) {
     Cell  *c = sheet[ i ][ j ];
     printf( "|%*s", getWidth( c ), "" );
   }
   printf( "|\n" );
 }
```



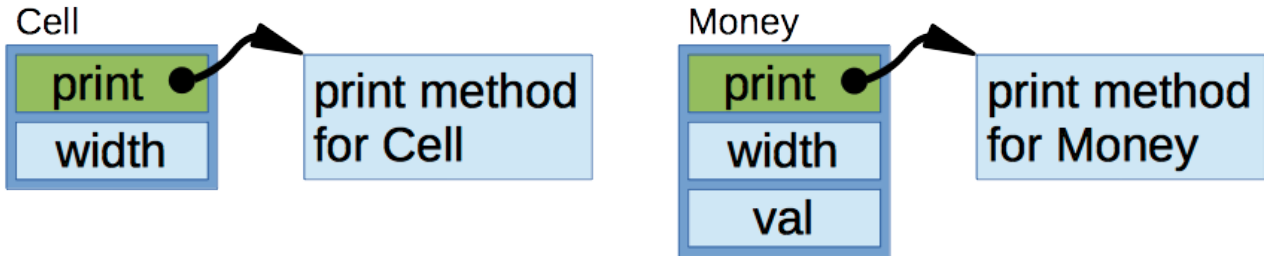
- If we had access restrictions, we could enforce encapsulation.

Overriding Methods

- What if we want these objects to know how to print themselves?
  - That's more interesting.
- Each type of object needs to print itself differently.
  - Cell : just print empty spaces
  - Money : print the val field as a monetary amount
- That sounds like overriding
  - A method that can behave differently in subclasses

Overriding Methods

- We can implement this with function pointers.
- The superclass can have a field for a pointer to the print method.
  - Which subclasses will also have.
- Each instance can carry around a pointer to the print method that's right for that instance.



- In C++, we would call this a *virtual method*.
  - We have to use a function pointer to get the right method.

Overriding Methods in C

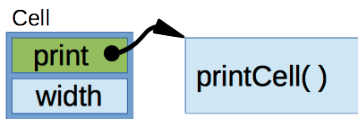
```
typedef struct CellTag {
  void (*print)( struct CellTag * );
  int width;
} Cell;

void printCell( Cell *this ) {
  printf( "%*s", this->width, "" );
}

Cell *makeCell( int width )
{
  Cell *this = (Cell *)malloc( sizeof( Cell ) );

  this->print = printCell;

  this->width = width;
  return this;
}
```



Structure for the superclass.

Print method for the superclass.

Store a pointer to the print method in the print field.

Overriding Methods in C

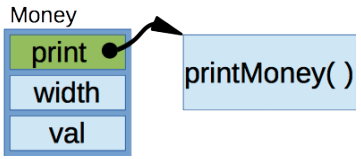
```
typedef struct {
  void (*print)( Cell * );
  int width;
  int val;
} Money;

void printMoney( Cell *cell ) {
  Money *this = (Money *)cell;
  printf( "%*d.%02d", this->width - 3,
          this->val / 100, this->val % 100 );
}

Money *makeMoney( int width, int val )
{
  Money *this = (Money *)malloc( sizeof( Money ) );

  this->print = printMoney;

  this->width = width;
  this->val = val;
  return this;
}
```



Structure for the subclass.

Print method for the subclass.

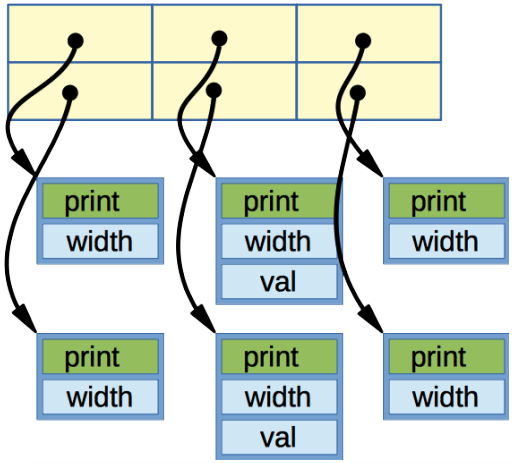
Store a pointer to the print method in the print field.

Calling the Print Method

- To call a virtual function like print, we use the pointer stored in the object's print field.

```
for ( int i = 0; i < 2; i++ ) {
  for ( int j = 0; j < 3; j++ ) {
    printf( "|" );

    Cell *c = sheet[ i ][ j ];
    c->print( c );
  }
  printf( "|\n" );
}
```



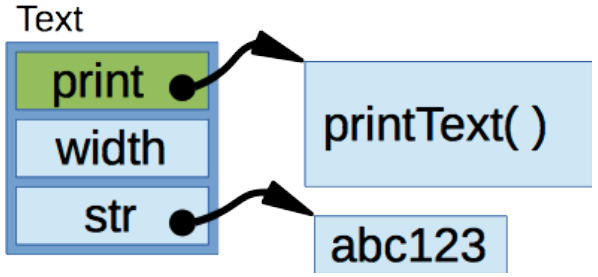

```
+----------+-----------+----------------+
|          |      3.75|                |
+----------+-----------+----------------+
|          |      8.90|                |
+----------+-----------+----------------+
```

Get the function pointer in this instance.

Pass in its this pointer, like any other method.

A More Complex Subclass

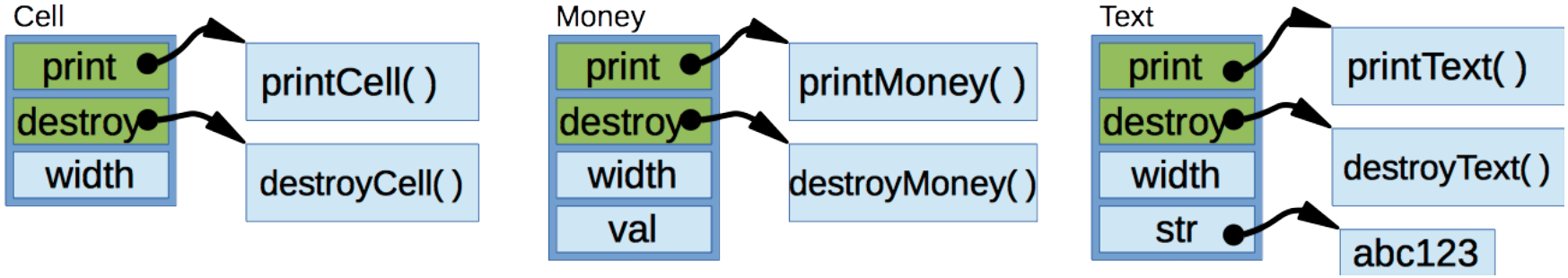
- Let's define another subclass, Text
- Stores an arbitrary, dynamically allocated string field



- Seems straightforward, but it reveals a problem
  - Freeing this object requires special behavior, freeing the dynamically allocated string.
- Solution: Cell and its subclasses should know how to clean up after themselves.

Virtualizing Destruction

- In C++, we will call this a virtual destructor
  - A function that knows how to free memory and other resources allocated by an object
  - The right function to use depends on the type of the object
- We can handle this with another function pointer in the superclass:



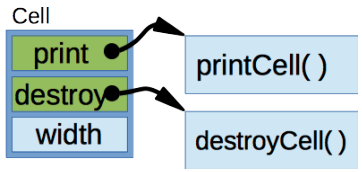
Virtual Destructor

```
typedef struct CellTag {
  void (*print)( struct CellTag * );
  void (*destroy)( struct CellTag * );
  int width;
} Cell;

...

void destroyCell( Cell *this ) {
  free( this );
}

Cell *makeCell( int width )
{
  ...;
  this->destroy = destroyCell;
  ...;
}
```



A new function pointer in the superclass.

A function to destroy a Cell (just free its memory)

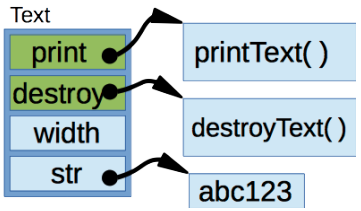
Store a pointer to the destroy method in the destroy field.

Text Objects

```
typedef struct {
  void (*print)( Cell * );
  void (*destroy)( Cell * );
  int width;
  char *str;
} Text;

void printText( Cell *cell ) {
  Text *this = (Text *)cell;
  printf( "%*s", this->width, this->str );
}

void destroyText( Cell *cell ) {
  Text *this = (Text *)cell;
  free( this->str );
  free( this );
}
```



Structure for text, with all the superclass stuff at the start.

A function to print a Text object.

A function to free a text object.

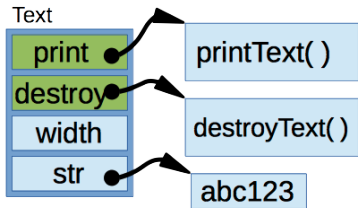
Text Objects

```
Text *makeText( int width, char const *str )
{
  Text *this = (Text *)malloc( sizeof( Text ) );

  this->print = printText;
  this->destroy = destroyText;

  this->width = width;
  this->str = (char *)malloc( strlen(str) + 1 );
  strcpy( this->str, str );

  return this;
}
```



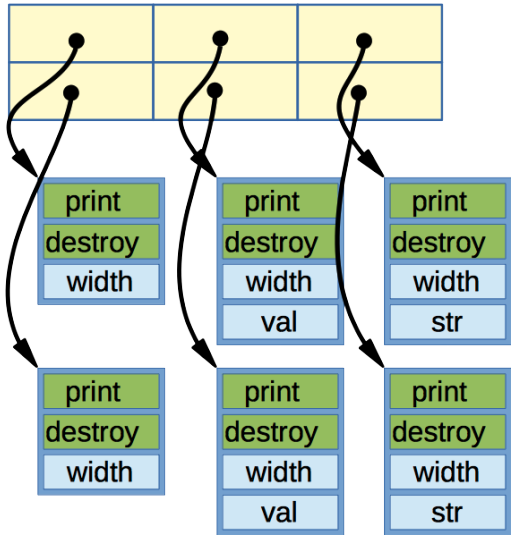
A function to create a text object.

Printing the Cells

- The same client code will print this new type of cell.

```
for ( int i = 0; i < 2; i++ ) {
  for ( int j = 0; j < 3; j++ ) {
    printf( "|" );

    Cell *c = sheet[ i ][ j ];
    c->print( c );
  }
  printf( "|\n" );
}
```




```
+----------+------------+-----------------+
|          |       3.75|              Gum|
+----------+------------+-----------------+
|          |       8.90|          Sandwich|
+----------+------------+-----------------+
```

Here, we added some text cells.

Freeing Cells

- And, each type of cell knows how to destroy itself

```
for ( int i = 0; i < 2; i++ ) {
  for ( int j = 0; j < 3; j++ ) {
    Cell *c = sheet[ i ][ j ];
    c->destroy( c );
  }
}
```