Pointers & Linked Lists

Gabe Johnson

Applied Data Structures & Algorithms

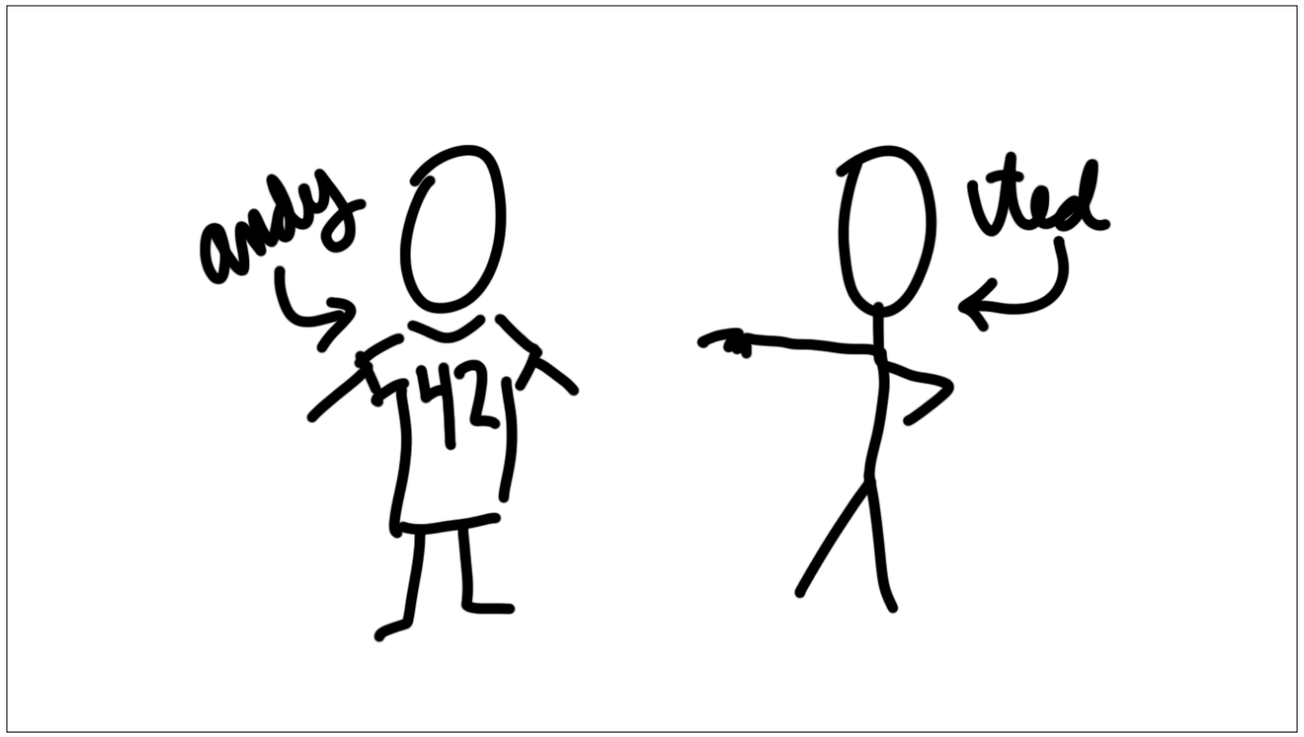
University of Colorado-Boulder

The first main topic we're going to dive into has to deal with how we store data in memory. Lists and linked lists are one of the simpler data structures, so we're going to start there. But before we talk about linked lists, we need to introduce a super important idea first: pointers.

Episode 1

Pointers

Pointers are one of the more brain-melty concepts that you're going to encounter in your computer science career. I hereby forgive you in advance if this is a hard topic to understand. That being said, just about everything that follows in this class will depend on you having an intuitive understanding of how pointers work.

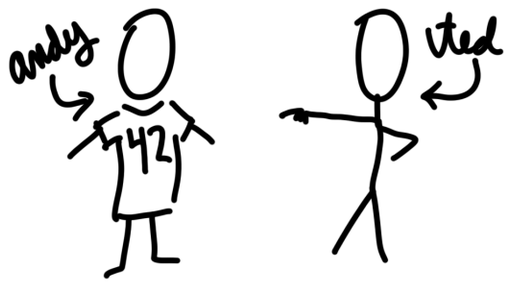


So maybe a silly picture will help introduce the topic.

This is Andy and Ted. Andy is wearing a shirt with a number on it. He has a numeric value, and that value is 42.

Ted is just standing there, pointing at Andy. His value is a reference to the spot where Ted is standing. We don't know what happened to Ted's shirt.

```cpp
#include <iostream>
using namespace std;

int main() {
  int andy = 42;
  int* ted = &andy;



}
```

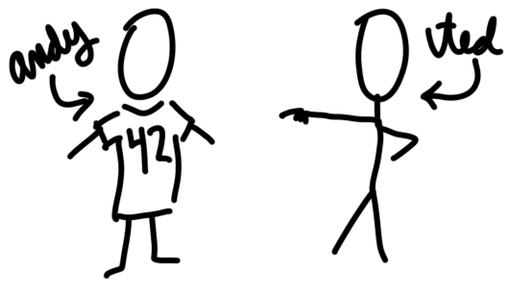


In C++, we can declare variables for both Ted and Andy using certain punctuation. Andy is easy, and you've seen this kind of thing before. int andy = 42;

But what's Ted doing? His role, or his TYPE, is to point at something. In this case, Ted is pointing at Andy. In C++ we declare a pointer variable with the type with an asterisk after it. In this case we want Ted to point to an integer, so we write: int* ted

That's the declaration. But we also want to assign a value. What’s Ted's value? Well, it's the spot where Andy is. We want to get Andy’s address in memory, and we do that using the ampersand in front of a variable.

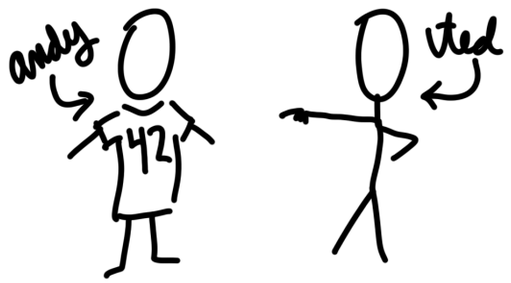
```cpp
#include <iostream>
using namespace std;

int main() {
  int andy = 42;
  int* ted = &andy;
  cout << " andy: " << andy  << endl;
  cout << "&andy: " << &andy << endl;
  cout << "  ted: " << ted   << endl;
  cout << " *ted: " << *ted  << endl;
}
```



If we were to print out all the stuff I just talked about, that code might look like this. First we get andy's value. Easy. Then we get Andy's address. Remember we put Andy's address in the Ted variable. So when we print out Ted, we should see the same thing as &andy.

Last, Ted with an asterisk in front means we're asking for the VALUE at the address that Ted is pointing to.

```cpp
#include <iostream>
using namespace std;

int main() {
  int andy = 42;
  int* ted = &andy;
  cout << " andy: " << andy  << endl;
  cout << "&andy: " << &andy << endl;
  cout << "  ted: " << ted   << endl;
  cout << " *ted: " << *ted  << endl;
}
```

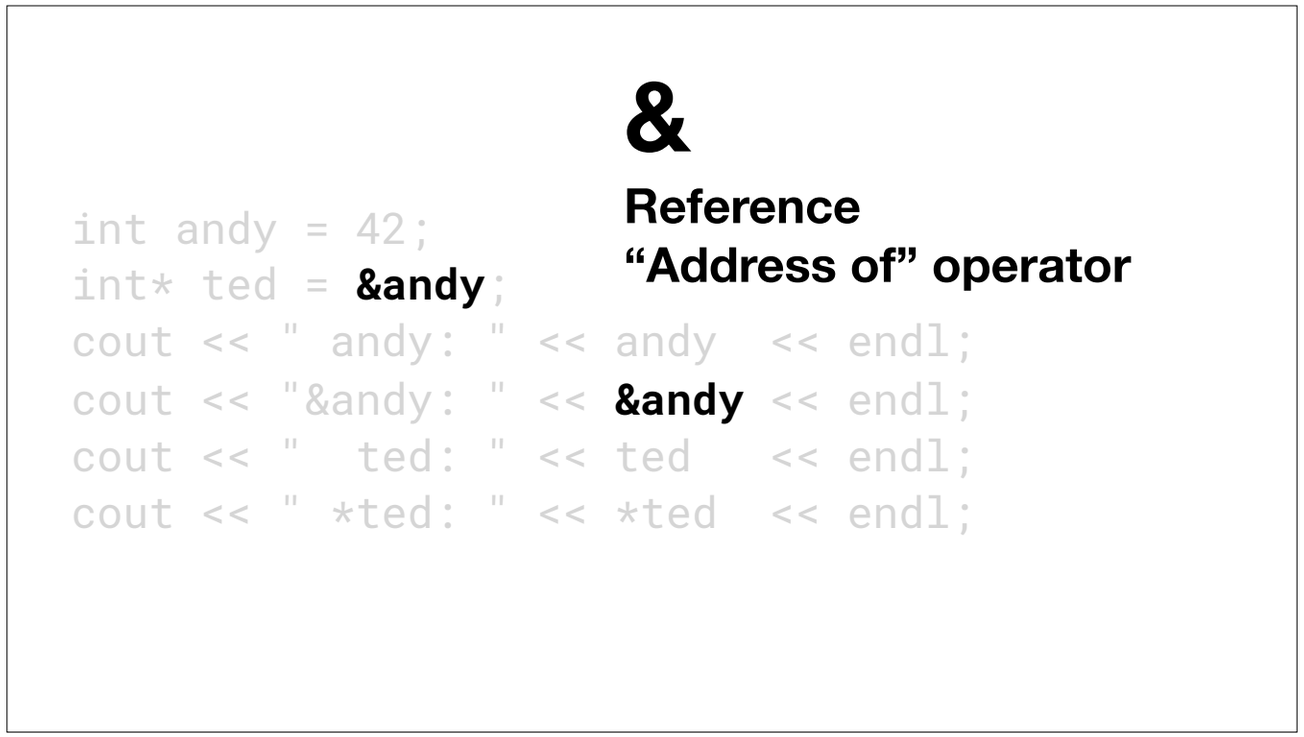



```
 andy: 42
&andy: 0x7fff508455bc
  ted: 0x7fff508455bc
 *ted: 42
```

And that's what we get. That gross 0x7ff thing is a memory address. That's just what it looks like when printed out. And importantly, both andy and star-ted give us 42.

```
int andy = 42;
int* ted = &andy;
cout << " andy: " << andy  << endl;
cout << "&andy: " << &andy << endl;
cout << "  ted: " << ted   << endl;
cout << " *ted: " << *ted  << endl;
```

I'm going to go over all of that again, calling out why we use all the confusing punctuation when we do.



First, the ampersand. Andy the ampersand! This is pronounced "reference to Andy" or, "address of Andy". Either way. When you stick it in front of a variable, you're telling the compiler to take the address of that variable, which gives you that icky 0x7ff thing that we just saw. Not very meaningful to humans but the compiler knows what to do with it.

```
int andy = 42;
int* ted = &andy;
cout << " andy: " << andy  << endl;
cout << "&andy: " << &andy << endl;
cout << "  ted: " << ted   << endl;
cout << " *ted: " << *ted  << endl;
```

*****

Dereference
“Value at address” operator

Next, the asterisk. Asterisk is actually used in two different (but related) senses. While the ampersand Andy was a reference, the asterisk Ted is a DEreference operator. When you stick an asterisk in front of a variable, you should pronounce it "value at address”.

int*

Pointer declaration
(this case is int pointer)

```
int andy = 42;
int* ted = &andy;
cout << " andy: " << andy  << endl;
cout << "&andy: " << &andy << endl;
cout << "  ted: " << ted   << endl;
cout << " *ted: " << *ted  << endl;
```

The other sense for using an asterisk is when we declare a pointer variable. Here we're using just integer pointers, and that's written int-star, or int with an asterisk after it. That tells the compiler that the variable points to an integer, so when you use the dereference operator, it knows you're talking about an integer.


```
int andy = 42;
```

```
int* ted = &andy;
```

In the following episodes and in the homework you'll see pointers a lot. It will start making more sense when you see them in action. But for now, keep that silly Andy/Ted cartoon in your head. Andy's an int, with a value of 42, Ted is a pointer to an int, and his value is Andy's address.

Episode 2

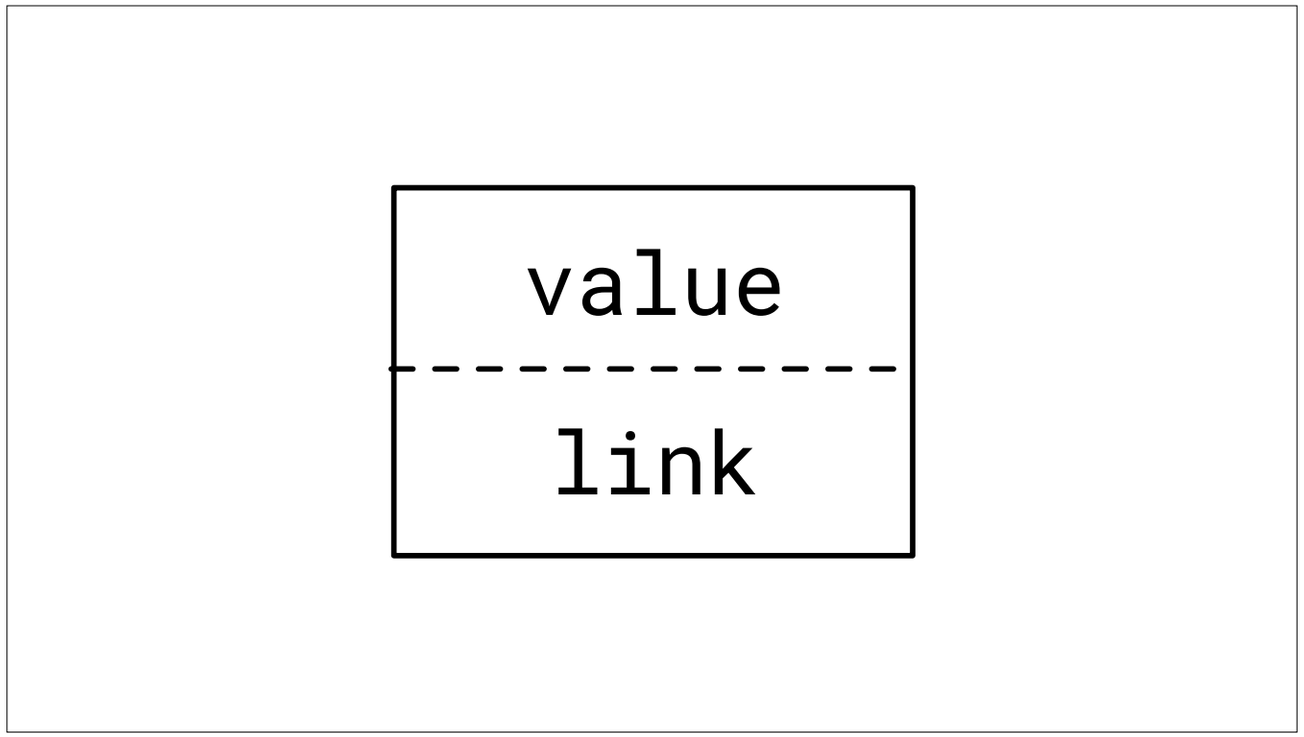
Linked Lists: Overview

Linked lists are a simple data structure that stores sequences of data. They're different from arrays in lots of ways. While we can still index into them, like "give me the 10th element of the list", we don't need to tell the compiler how many items we plan to put in the list when we declare it. Because... maybe we don't know how big it needs to be? Linked lists grow (or shrink) depending on how we're using them. And one of the tools we're using to get that effect is by dynamically grabbing memory and storing pointers to that memory.

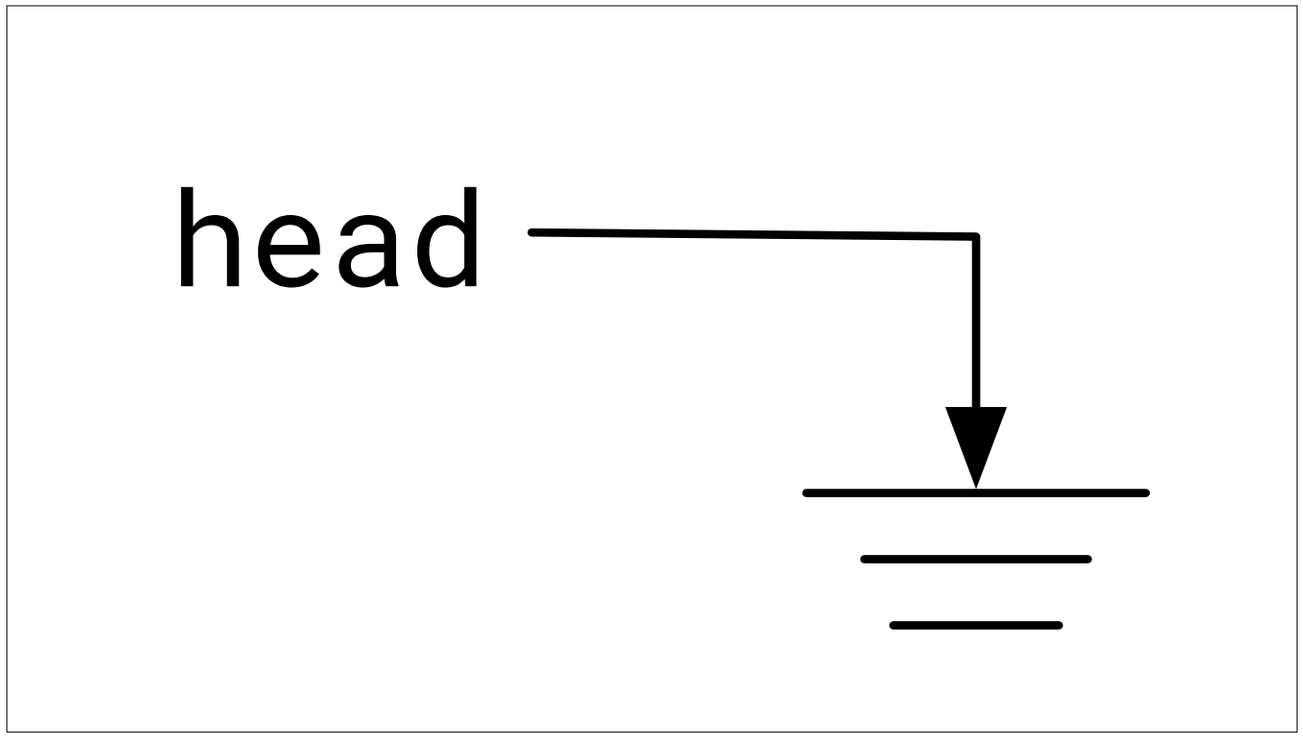
```
struct node {
   int data;
   node* next;
};
```

A linked list is made up out of node structures, each of which has a value, and a link to another node. In C++ we can define a node like this using the struct keyword.

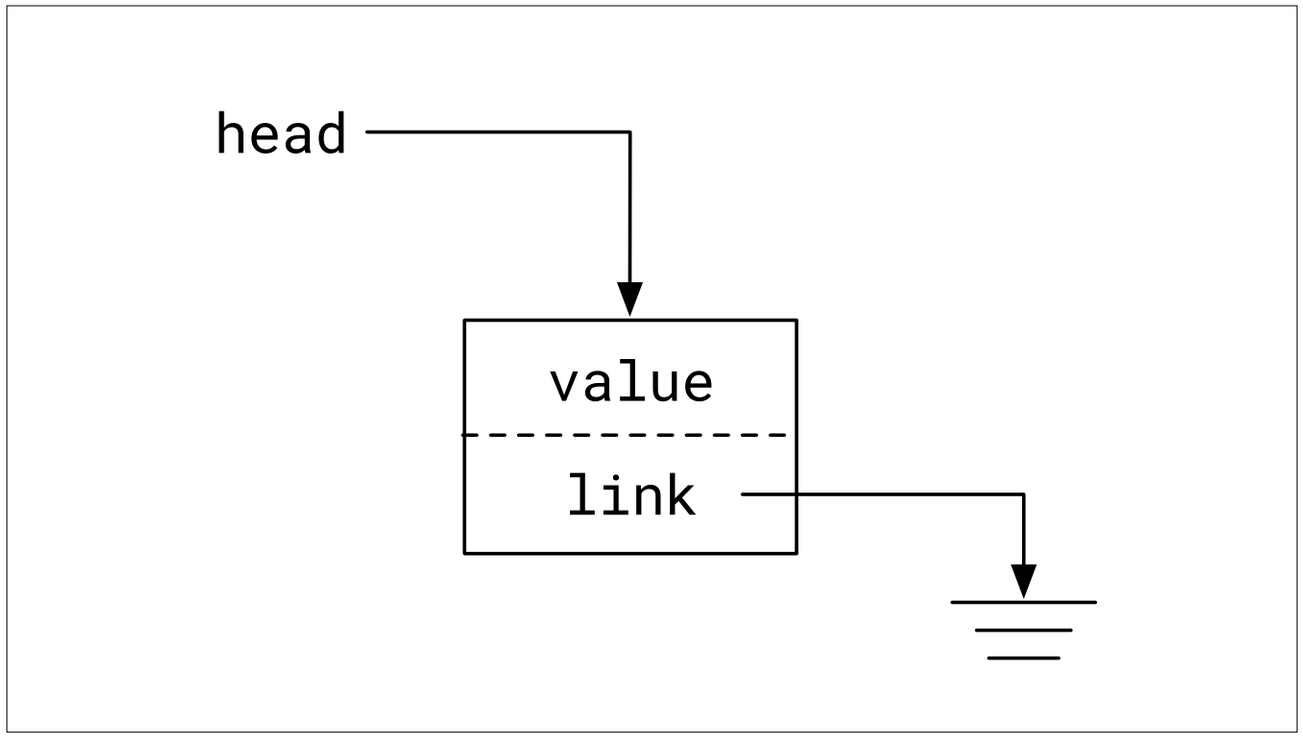
This linked list node can only hold integers, but we could make different structs that have text or whatever else inside of them. And other than the data, there's this node* thing called next. That's declaring a field in the structure, and its type is node star, which is a pointer to a node. You'll find throughout this class that making diagrams will help immensely.



Here's how I draw a linked list node.



To have a list, you need a pointer to the first node. Let's call it ‘head'. Each node's next field points either to the next node in the sequence, or if it is the last item, it points to NULL. The empty list is drawn like this.

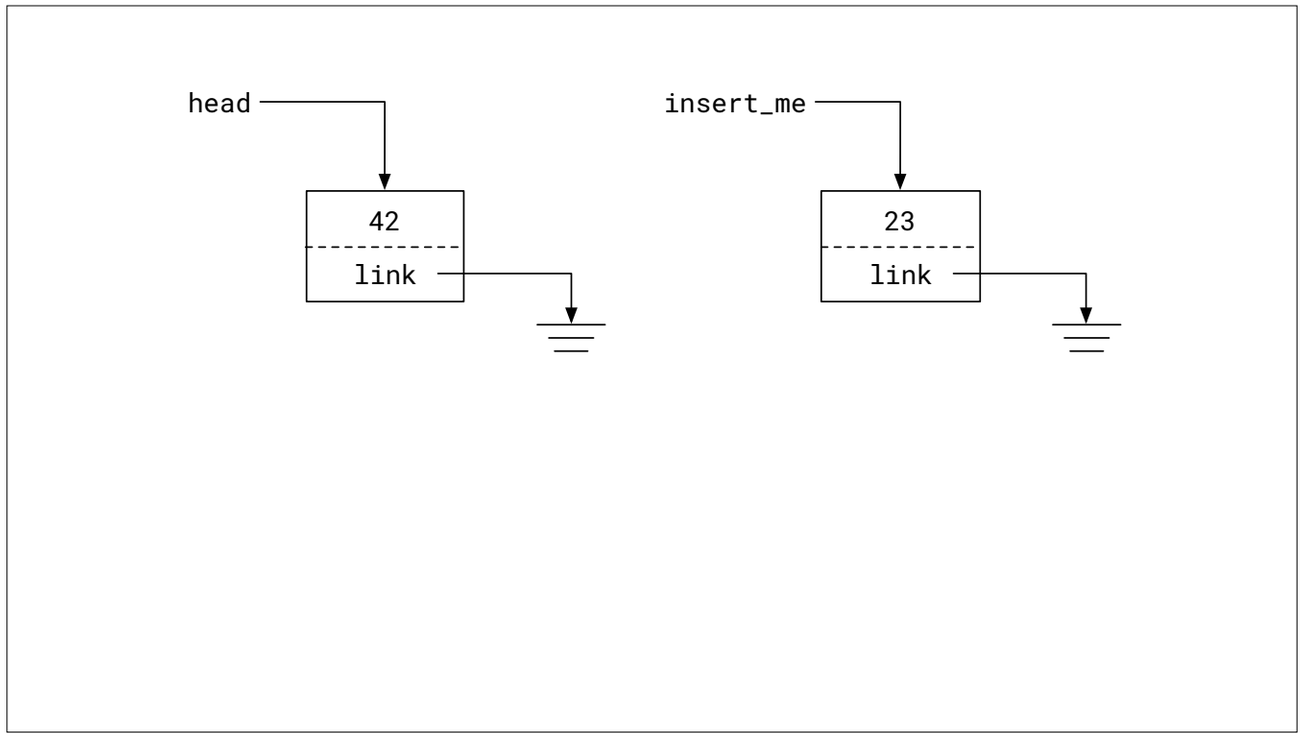


Here's a list with one node in it:

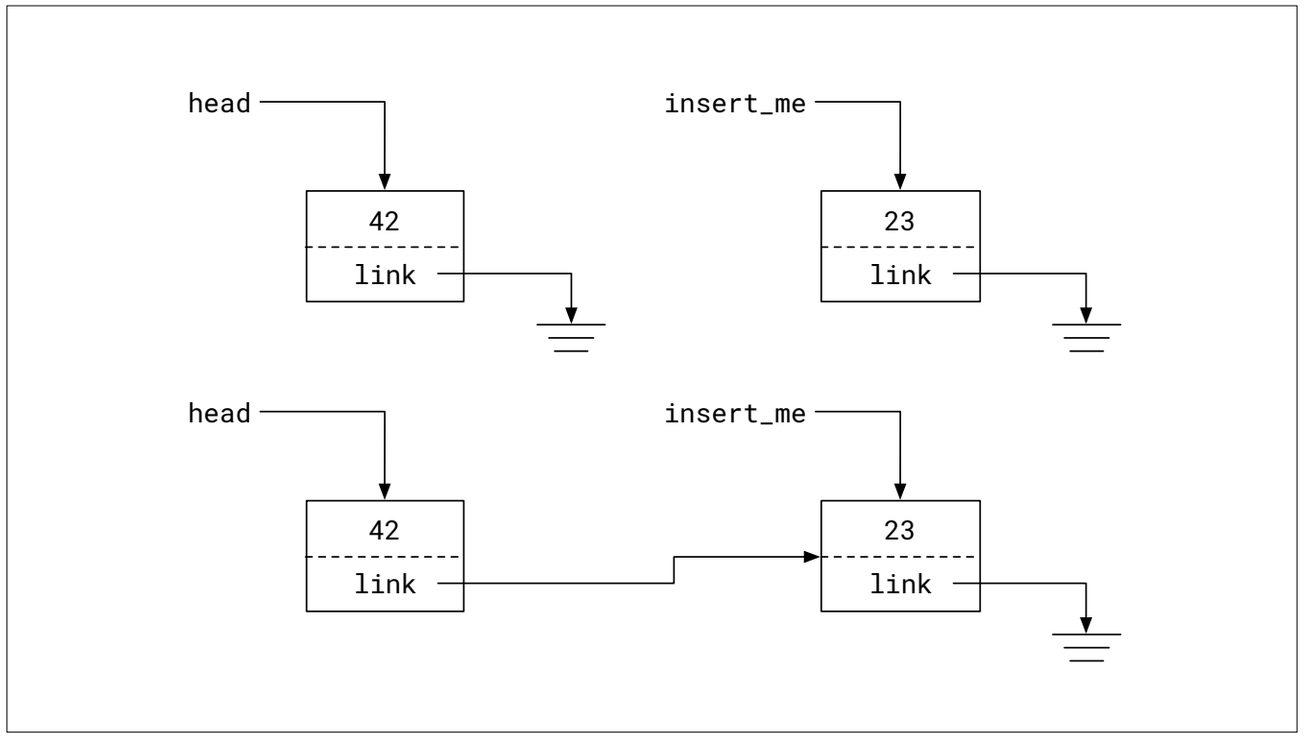
Data Structures Have Operations

- make a new thing
- add to the thing
- remove from the thing
- query the thing
- and other amazing things

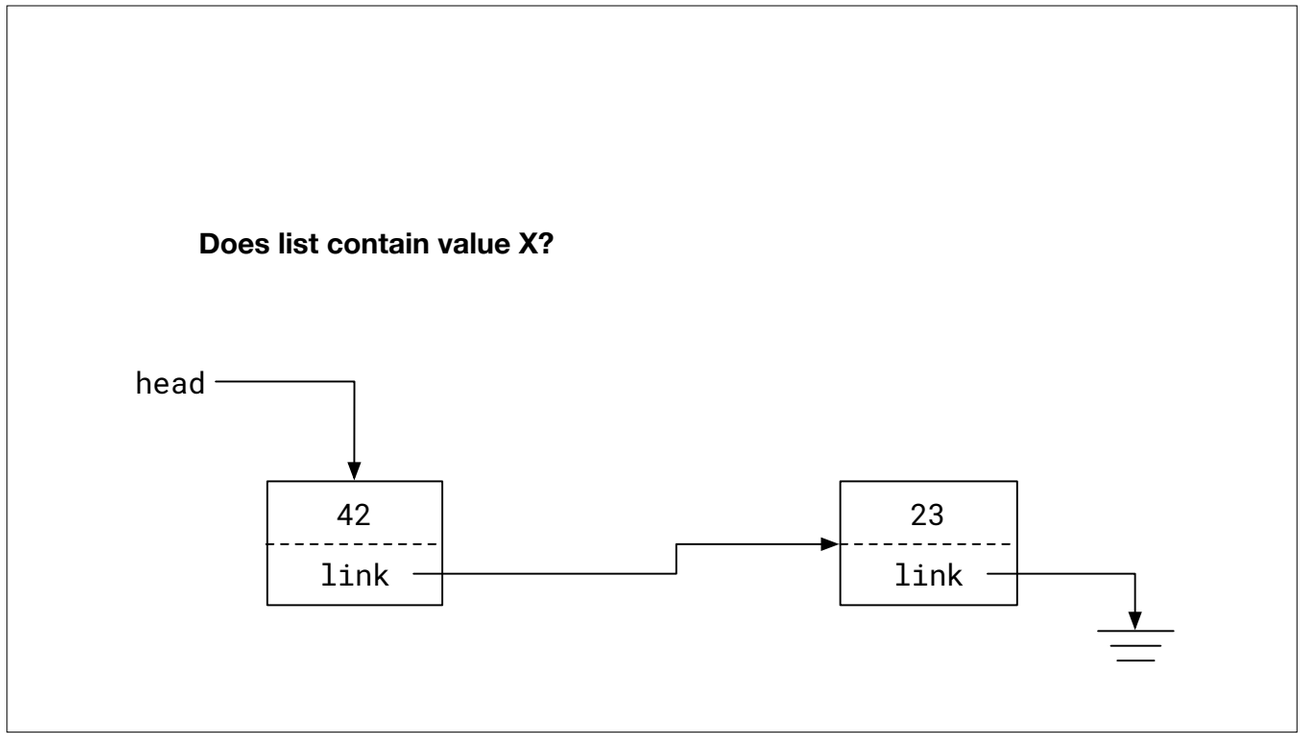
Datastructures just about always have a set of operations that go along with them. These operations let us do things to the datastructure, like adding and removing data, or let us query the datastructure, or telling us if it contains something.



One of the linked list operations is append. Say we have a list with one item, 42, in it, and we want to add 23 to the end. So the list pointed to by top has 42, and this new list is pointed to by 'insert_me'. To append this new item to the existing list, we have to write code that transforms this situation into…

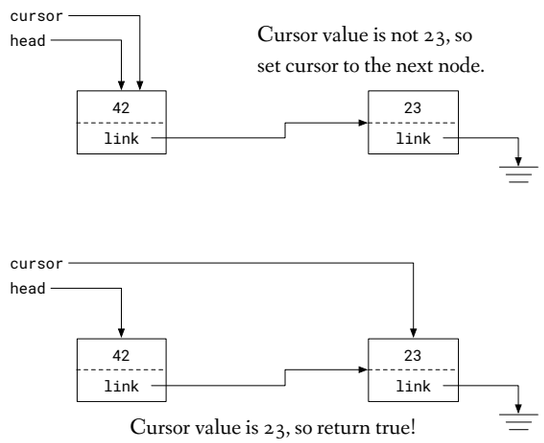


this. My point here is that drawing a picture will really help you figure out what you need to pay attention to. Of all the values and pointers, what has changed between the upper and lower pictures?



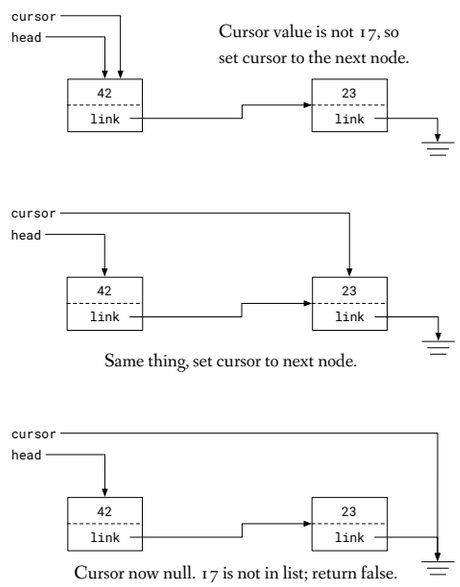
Another operation is to query a list to see if it contains a given value. To do this in code, we have to start at the first node and examine each node for the target value. If we find the value, return true. But if we get to the end (because the next node is null), we know the list doesn't have it and we return false.

Looking for 23…



I'll illustrate that. Say we're looking for 23. We'll use a cursor variable that points to the node we're inspecting right now. At first, set the cursor to the top node. What's the value? 42. Not what we're looking for, so update the cursor using cursor's next field. Now when we look at cursor's data, we get a match. Return true!

Looking for 17…



But say we are looking for a number that's not in the list, like 17. We'd do the same process as before, setting a cursor to point at the first node, then the second. Then we can't go any further because the last node's next field points to null. That's our signal that we've reached the end of the list, so we return false.

Linked List Operations

- initialize
- append
- insert
- remove
- query
- size
- report

The operations we're going to implement on linked lists are: initialize (make a new linked list), append (add an item to the end), insert (add an item at some index), remove an item at some index, query a list to see if it contains an item, a size function to tell you how many items the list has, and a report function that gives you the contents of the entire list as a string. We'll spend an entire episode talking about these. But first, we need to talk about memory.

Episode 3

Stack and Heap Memory

```
void someFunction() {
  int x = 4; // value 4 stored in stack memory
  int* y = new int(6); // value 6 stored in heap memory
}
```

This episode is about two different uses of memory, namely Stack and Heap. It is important to understand the difference between these, and when and how to use each. This function shows both uses. The first line uses stack memory, and the second uses heap memory.

```
int timesTen(int y) {
  int x = 10;   // x uses stack memory
  return y * x;
  // stack memory automatically reclaimed
  // when function exits
}
```

When you write a C++ function, you'll write things like `int x = 10;` or whatever. And when your function ends, all the variables that you declared like that are reclaimed by the operating system so it can be re-used. Variables that have this life cycle are called stack variables because the memory they are recorded in is called "the stack".

```
void eatMemory() {
  int* y = new int(10); // y points to heap memory
  cout << " y: " << y  << endl;
  cout << "*y: " << *y << endl;
  // heap memory is not automatically reclaimed
  // when function exits. memory leak!
}
```

Sometimes you'll use the 'new' keyword when assigning new variables, like `int* y = new int(10);`. The `new int(10)` part allocates memory from a different kind of memory called "the heap", and returns a pointer to it. And the `int* y` part on the left hand side allocates memory from the stack to store that pointer. When a function ends, stack variables, like the one storing the int pointer y, are reclaimed. But the memory it points to, the spot with the value 10, will persist until it is recycled with the `delete` keyword.

Stack vs Heap Memory

Stack Memory:

- **Short life cycle**
- **Easy to take care of**
- **Reclaimed after function returns**
- **Not persistent**

So: stack variables have short life cycles. They're easy to take care of. But they disappear when the function returns. So if you need data to persist beyond what the stack variable life cycle gives you, you need to use a heap variable.

Stack vs Heap Memory

Stack Memory:

- **Short life cycle**
- **Easy to take care of**
- **Reclaimed after function returns**
- **Not persistent**

Heap Memory:

- **Can have long life cycle**
- **Use ‘new’ to create data on heap**
- **Programmer must manage memory**
- **Use ‘delete’ to free that data**
- **Can lead to memory leaks**

Like I mentioned sort of obliquely earlier, data on the heap is allocated when you use the `new` keyword, and it will stay around until it is freed with the `delete` keyword. And if you the programmer don't get around to freeing that memory, you'll end up with what they call a 'memory leak'. That's when your program has claimed all this memory for some past purpose but hasn't recycled it. Eventually, if your program lasts long enough, you'll start running out of memory, and the whole computer starts running slower.

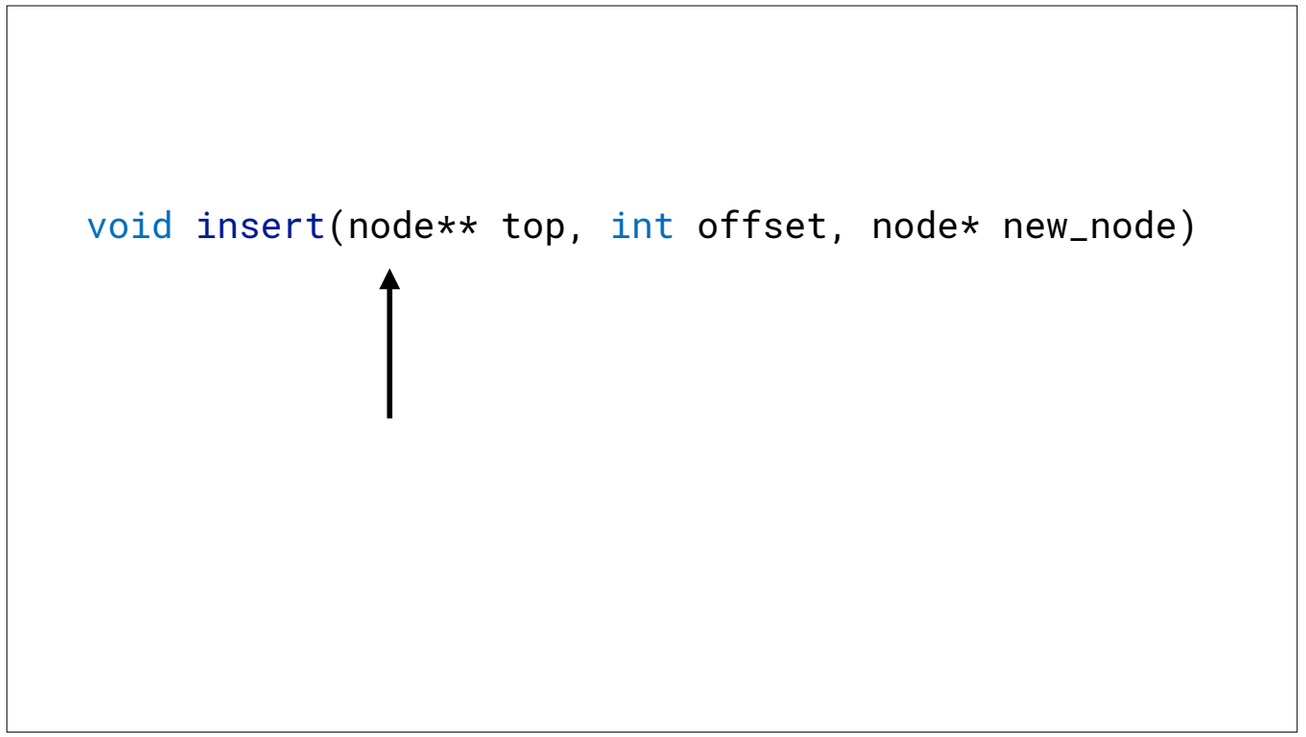
```
void dontEatMemory() {
  int* y = new int(10); // y points to heap memory
  // ...
  delete y;
}
```

By the way, here's the delete keyword in action. If you have a pointer type y, just say delete y, and it frees the memory it points to.

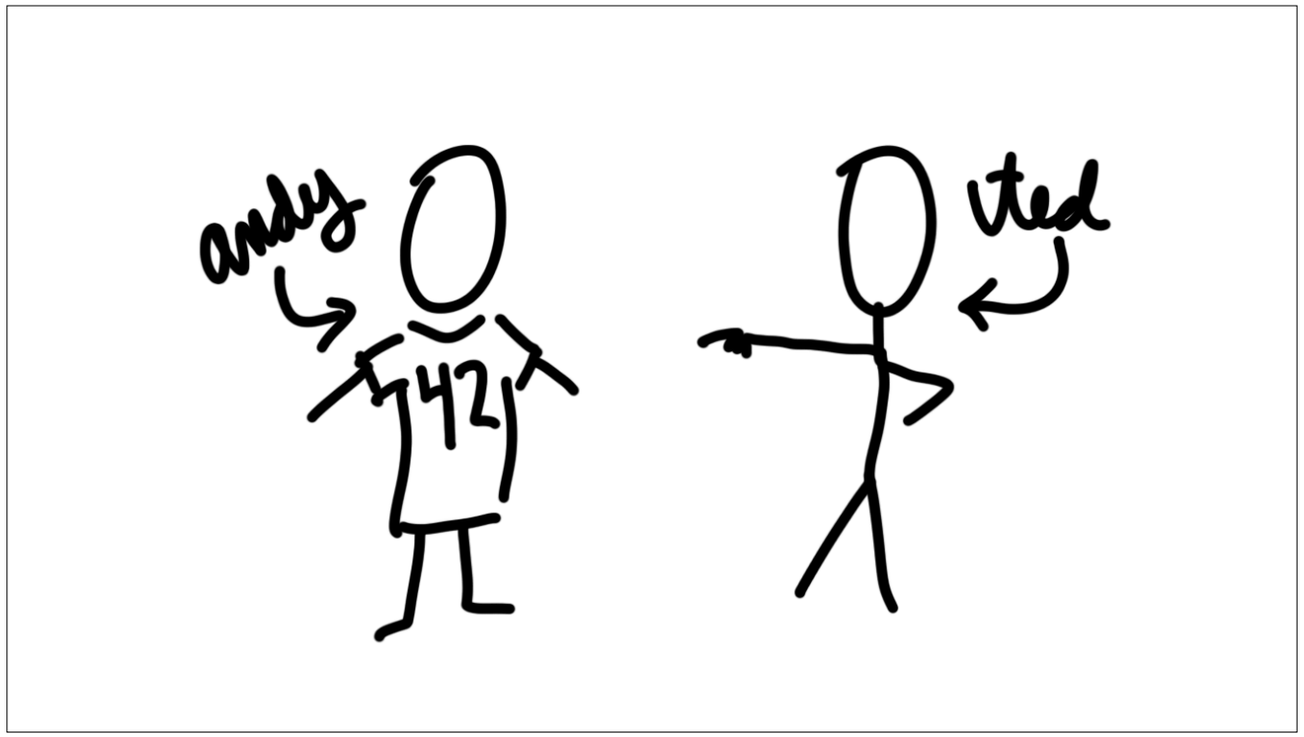
Episode 4

Double Pointers

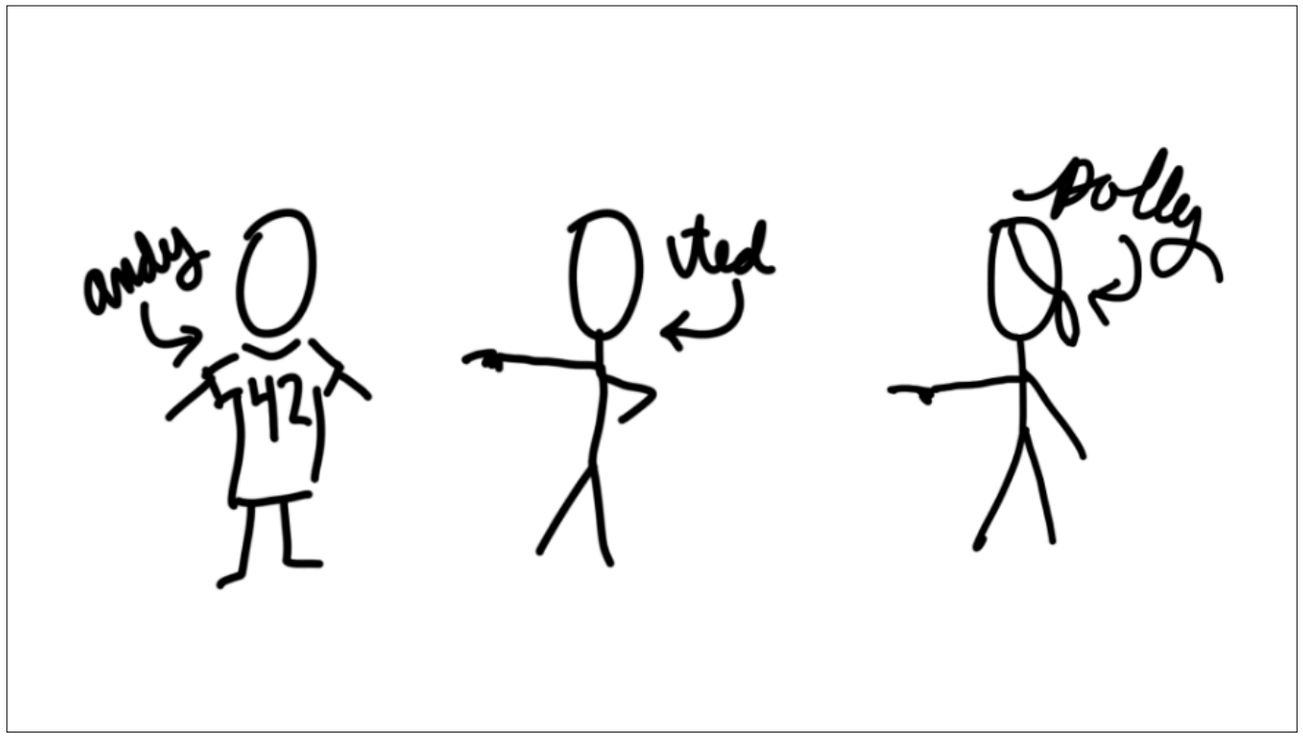
In this episode we’re talking about double pointers, for double the brain melty fun!



If you've looked at the linked list homework assignment, you might have seen the function signatures that you're going to implement. If you looked at the homework and saw this one… your head might have exploded a little bit with that node-star-star in there. Don't worry. This episode aims to clear this up.



Let's go back to Andy and Ted.



And their new friend, Pointing Polly. Polly is a double pointer. She's pointing to Ted, who is pointing to Andy.

```cpp
#include <iostream>
using namespace std;

int main() {
  int    andy = 42;
  int*    ted = &andy;
  int** polly = &ted;




}
```

A double pointer is declared just like a single pointer, but there are two asterisks. And yeah, you can have triple and quadruple pointers, and there are reasons to have them, but personally I'd consider a career change if I had to deal with them.

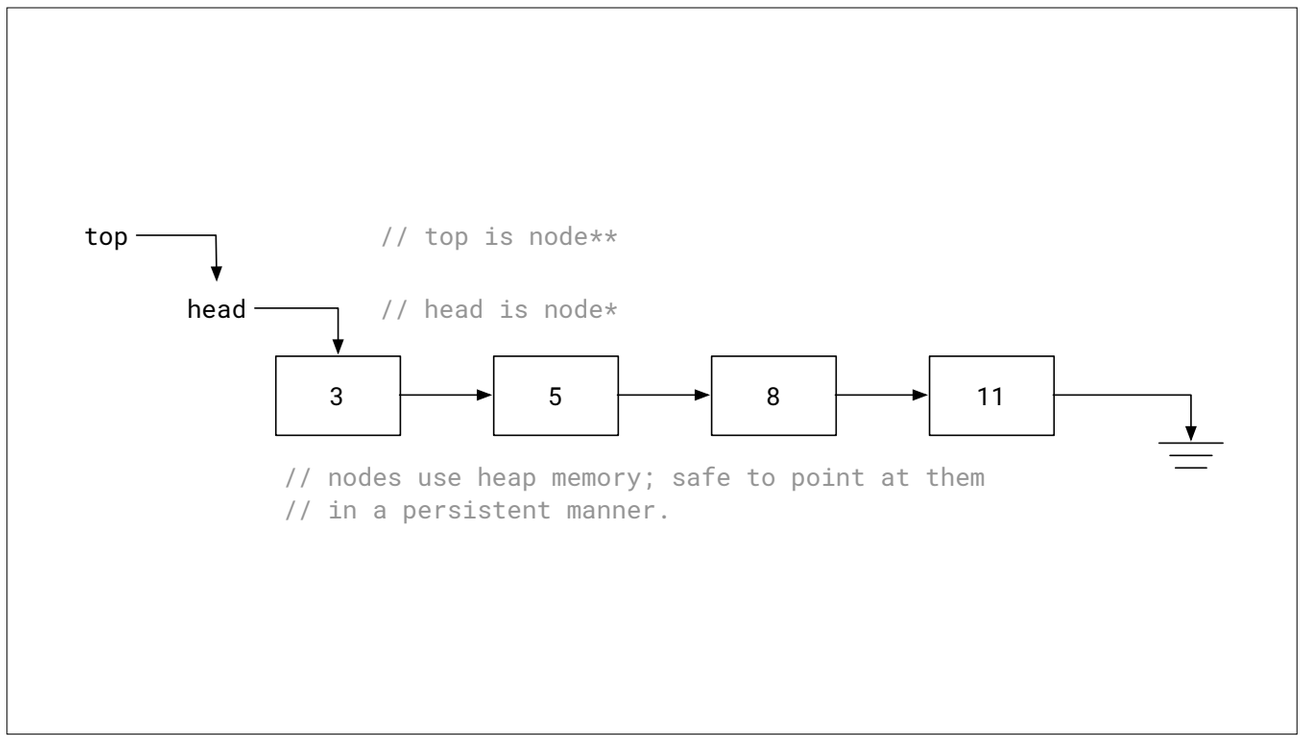
```cpp
#include <iostream>
using namespace std;

int main() {
  int    andy = 42;
  int*    ted = &andy;
  int** polly = &ted;

  cout << "   andy: " << andy << endl;
  cout << "  &andy: " << &andy << endl;
  cout << "    ted: " << ted << endl;
  cout << "   *ted: " << *ted << endl;
  cout << "   &ted: " << &ted << endl;
  cout << "  polly: " << polly << endl;
  cout << " *polly: " << *polly << endl;
  cout << "**polly: " << **polly << endl;
}
```

Anyhow, printing out the values and addresses and such like earlier looks like this. Stare at that for a second, and ask yourself what do you expect the last two lines to print out? What's *polly going to be? She's pointing at Ted, so that should print out Ted's address. And star-star-polly? What's that mean?

```cpp
#include <iostream>
using namespace std;

int main() {
  int    andy = 42;
  int*    ted = &andy;
  int** polly = &ted;

  cout << "   andy: " << andy << endl;
  cout << "  &andy: " << &andy << endl;
  cout << "    ted: " << ted << endl;
  cout << "   *ted: " << *ted << endl;
  cout << "   &ted: " << &ted << endl;
  cout << "  polly: " << polly << endl;
  cout << " *polly: " << *polly << endl;
  cout << "**polly: " << **polly << endl;
}
```


```
   andy: 42
  &andy: 0x7fff580d655c
    ted: 0x7fff580d655c
   *ted: 42
   &ted: 0x7fff580d6550
  polly: 0x7fff580d6550
 *polly: 0x7fff580d655c
**polly: 42
```

Well, here it is. star-polly is indeed Ted's address, and star-star-polly is 42!



We need to be able to refer to a linked list for potentially a long time, and outside of the function that created it. So in order to do that, we need to use heap memory for our list data.

So if a `node*` variable points to the first node in a linked list, what happens when we want to do an operation that should change _which_ node is the first one? That's why we have the `node**` variable in the linked list homework. We not only need to keep track of nodes using pointers, but we also need to keep track of which of those pointers references the first element in the list, and to do that we have to point to a pointer.

When you pass in a node star star to a function, you're basically handing that function a slip of paper that it will write down the new address of the first node. When the function returns, the calling context can look at the paper to know where the first node is now.

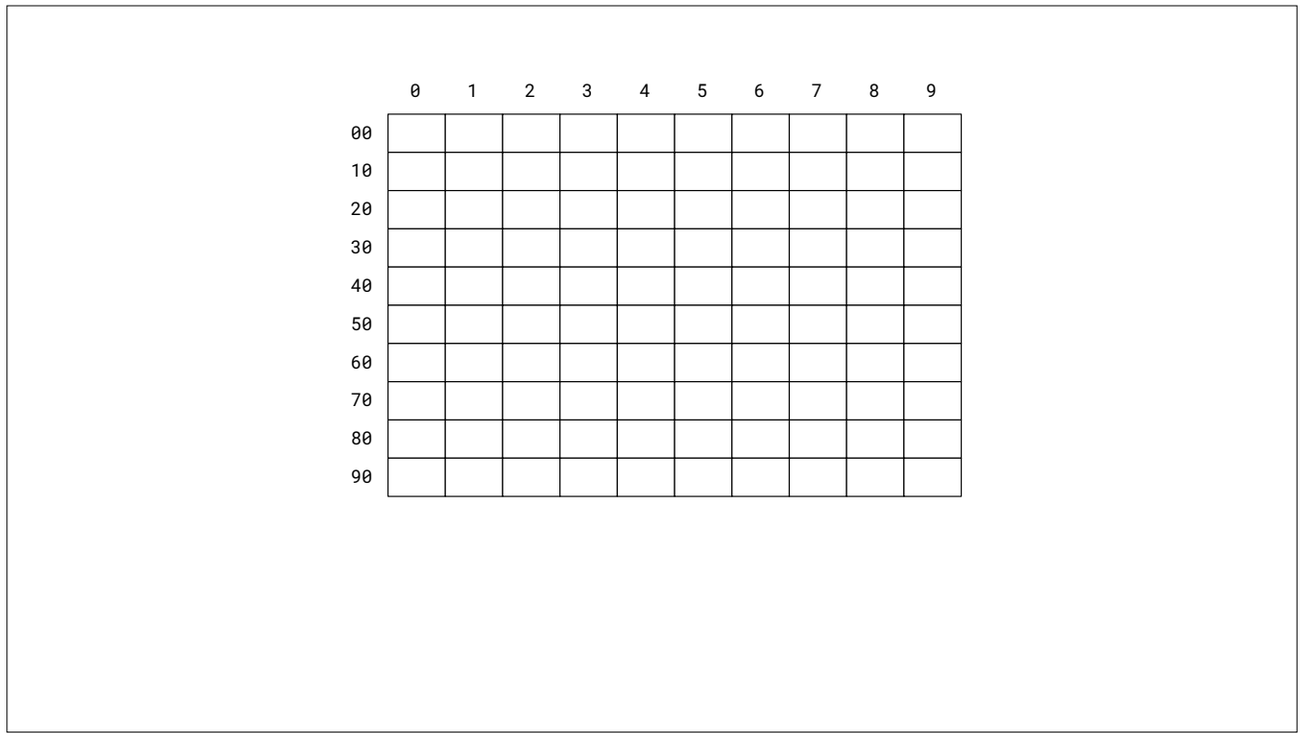
```cpp
// naming conventions and hints for the homework assignment
node* head = init_node(1); // head = pointer to first node in list
node** top = &head;        // top  = pointer to head
*top = head;               // this is how to assign first node
cout << "value of first node: " << (*top)->value << endl; // prints 1

node* cursor = head;       // cursor = pointer to any node
while (cursor != NULL) {   // loop as long as cursor isn't NULL
                           // do something great here.
  cursor = cursor->next;   // be sure to update cursor!
}
```

By the way. We're trying to stick to a naming convention here with the double pointers and single pointers. We'll use `top` for the double pointers, because it sits above everything else and is used to access whichever node is the first one at the moment. We'll use `head` for the single pointer to the first element, also known as the head of the list. And in the homework we will also use `cursor` as a single pointer, and that's used to scroll through our list.

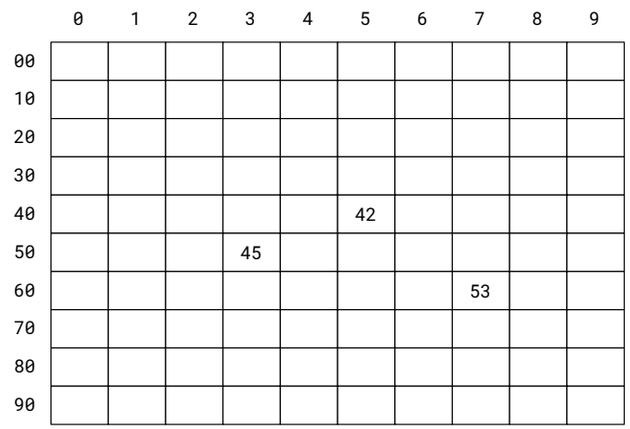
Episode 5

Visualizing Memory Usage

In this episode we'll see a useful technique for visualizing what's going on with memory when we use pointers.



This is Tiny Memory Land, which is a hypothetical computer memory that has 100 memory addresses, each of which can store either an integer, or a pointer to an integer. When new memory is needed, the hypothetical computer picks a free address at random. Let me stress here with an awkward pause... this is not how real computers work, but for the purposes of describing pointers and stack and heap memory, it is a useful simplification.



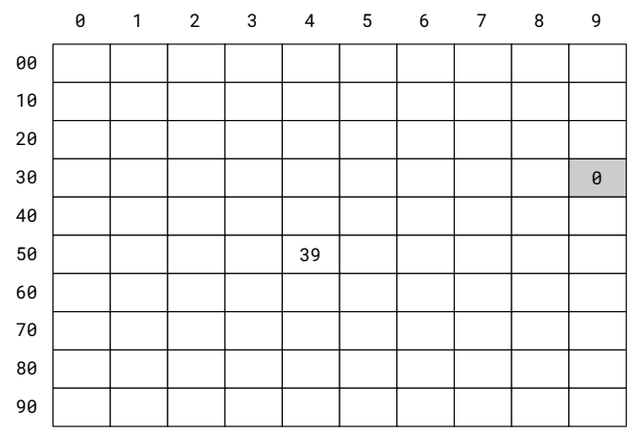

```
int   andy  = 42;     // andy is given address 45
int*   ted  = &andy;  // ted is given address 53
int** polly = &ted;   // polly is given address 67
```

Let's visualize this Andy / Ted / Polly example. On the first line, we say "give me a new stack variable for an integer named Andy, and put the integer 42 in it." In our Tiny Memory computer, it picks a free memory cell at random, which just happens to be address 45. Then on the second line, we say "give me a new stack variable for a pointer named Ted, and put Andy's address in it." Tiny Memory computer picks another free cell at random, which just happens to be address 53. And then on the last line, I hope I'm not beleaguering the point, we say "give me a new stack variable for a pointer named Polly, and put Ted's address in it." Tiny Memory picks cell 67 at random and puts Ted's address in it. In this example we're only using stack variables. But what if we want to manipulate heap variables?

```cpp
void heapMem() {
  int* target = new int(0); // target is a stack var that points to heap memory
  cout << "Before calling populate:" << endl;
  cout << " target: " << target << endl;
  cout << "*target: " << *target << endl;
  populate(target);
  cout << "After calling populate:" << endl;
  cout << " target: " << target << endl;
  cout << "*target: " << *target << endl;
}

void populate(int* where) {
  *where = 87;
}
```

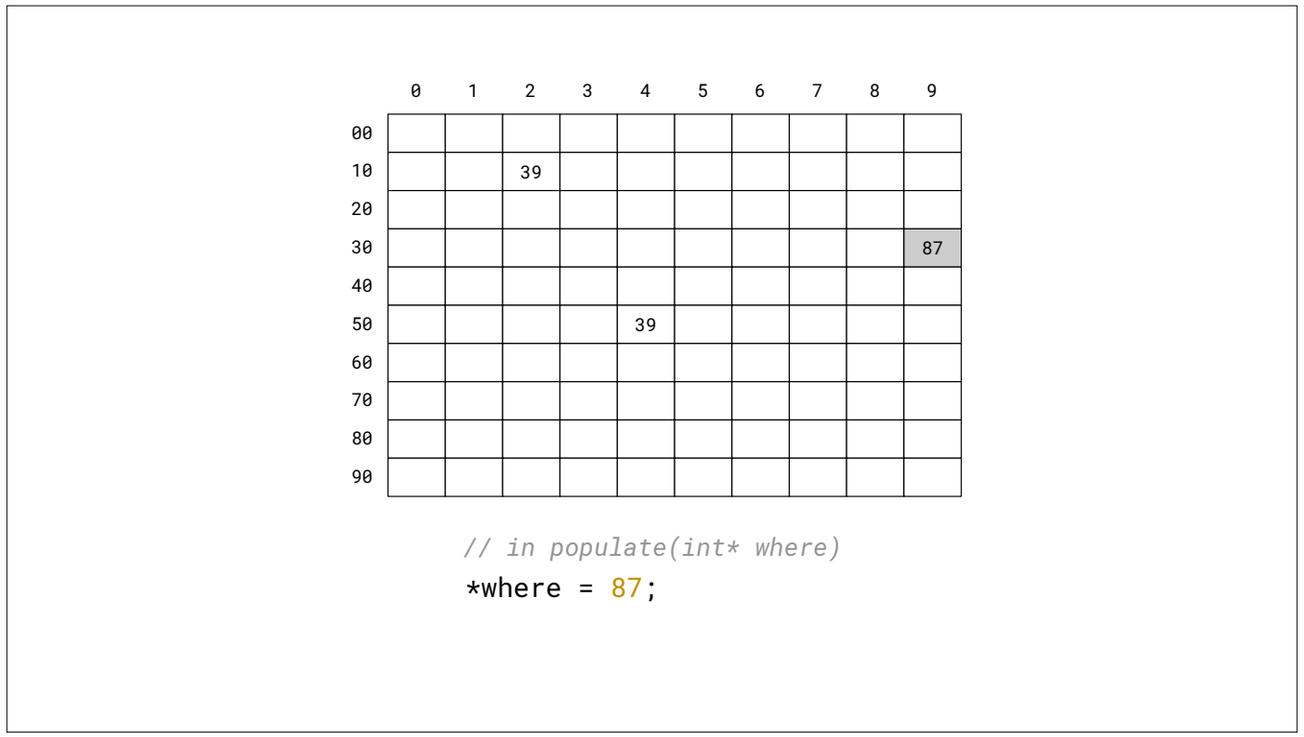
```
Actual C++ output:

Before calling populate:
 target: 0x7fdee5402700
*target: 0
After calling populate:
 target: 0x7fdee5402700
*target: 87
```

This example shows us allocating heap memory and passing a reference to that memory to a `populate` function that writes to the memory location we're passing in. It seems simple but there's a load of important stuff going on here. First line, "give me a new stack variable named target that points at a heap memory location for an integer." I'll show pictures in a second. We print that out, and the actual C++ on my computer shows that target is a memory location, and star target is zero.

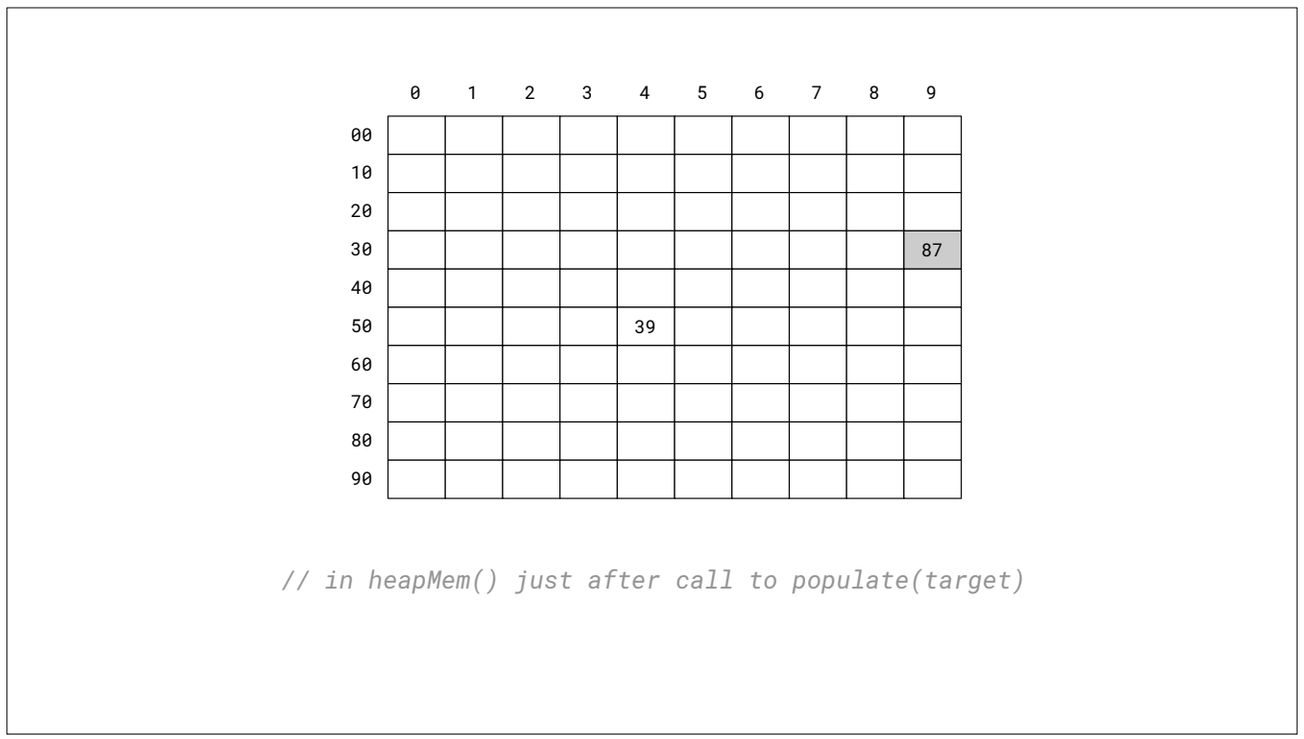



```
// in heapMem() at the top
int* target = new int; // target is given address 54
```

Here is what the Tiny Memory view of this might be. That same line where we declare `target` and give it a value actually involves two different memory locations: one is the heap memory we're allocating, and that's shaded, and the other is the stack variable to hold the pointer, and that's not shaded. Our stack pointer happens to be at location 54.



When we issue the call to `populate`, we give it our pointer as an argument. But check this out! When our populate function executes, it gets a copy of the inputs. So we grab a free cell from stack memory for our `where` variable, and the value of `target` is copied into there. So now our tiny memory has two pointers, one is a stack var in the first function, and another stack var in the second function, but both point to the same spot. This drawing shows tiny memory just after executing the one and only line in the populate function: `*where = 87`.



After the `populate` function returns, all of its stack variables are freed up. So this is what it looks like in the original function after the call to populate has returned, and you'll notice there's now only the one pointer, `target`. So when we print out the value of where it points to, you can see that the heap memory now has 87 written in it.

Episode 6

Contracts

This episode is about code contracts, which is a strategy for planning what you’re about to do. This is the last of the tangential videos before we get back to linked lists.

Whenever you implement a function, it is a good practice to think about it in terms of inputs, outputs, and what are the basic rules for what our function can and can't do. There is a style of programming called design by contract, and it can help you and others work together by defining behaviors and providing guidance about who is responsible for what.

Code Contracts

Pre-Condition

What can we rely on as being true when we begin?

Post-Condition

What can we rely on as being true when we finish?

Invariant

What requirements for program state must we guarantee are true?

The rules for inputs are called "pre-conditions". What are the requirements on the arguments passed in? Like can numbers be negative? Or zero? Or is there an upper limit?

Similarly the rules for outputs are called "post-conditions". And when the function exits, what are the legal states you can be in? Do you want your function to have side effects? Or is it a wholly-contained black box whose only effect is to return a value?

And there's another related word, "invariant", which is a rule that your code must obey.

Non-code example

Here's a non-code example.

Susie, please take this empty jar over to Jack's house, fill it with sweet delicious curry, and bring it home. Try not to burn the house down.

Precondition: Susie brings empty jar to Jack's house.

Precondition: Susie brings an empty jar to Jack's house. She can't go without a jar, and she can't go with a full jar. Must be a single empty jar. She can't go to Polly's house. Must be Jack's house.

Postcondition: Susie brings home a jar full of sweet delicious curry.

Postcondition: Susie returns home with a jar full of curry. The curry must be both sweet and delicious.

Invariant: house must not burn down.

Invariant: At no point in time must the house burn down.

Pre-condition

```
double root(double* x);
```


The calling context is responsible for making sure that x has a valid value.

Pre-condition is the definition of what's valid.

Say we are implementing a function that computes a real square root (no imaginary numbers here). We're giving some number `x`, and we return the square root of `x`. What possible values for `x` can we expect? For starters, negative numbers aren't allowed, since we aren't allowing imaginary results. What about zero? I had to look this one up because I'm bad at math. That's allowed, and the result is zero.

Say your boss demands that the function signature has to look like this, with a pointer to a double. Set aside for the moment the fact that this is crazy. For another precondition, we might require the input pointer is not NULL.

Typically it is up to the calling context to provide inputs that obey the preconditions. This makes it much easier to implement, since we don't have to do things like checking to see if the input value is negative, or NULL. If we can't depend on the calling context to provide good inputs, then we might have to check those inputs ourselves.

If the function is going to check input, then you'll need to define exactly how you'll handle unexpected or otherwise bad input. And that's related to postconditions.

Post-condition

```
double root(double* x);
```

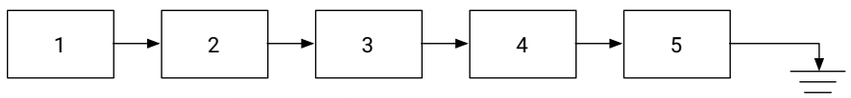

The root function is responsible for making sure that it returns the correct value and leaves the program in an acceptable state.

Post-condition is the definition of what's correct and acceptable.

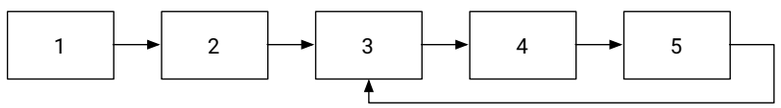
A function's job is to guarantee that it meets its postconditions. For our silly `root` function, we might have one postcondition that we've returned the square root of the input value, and furthermore that we haven't manipulated the input in any way. Since it is a pointer, we could really mess things up by changing its value.

Invariants

```
// invariant: linked lists must terminate with NULL
```




```
// ok: 1, 2, 3, 4, 5, done.
```




```
// not ok because it never terminates.
// 1, 2, 3, 4, 5, 3, 4, 5, 3, 4, 5, 3, 4, 5, etc.
```

An invariant is a rule that we must obey. For example with our linked lists, we might have an invariant that every list terminates with a null pointer. Because otherwise we could have a linked list that never ends! And there are similar data structures for which that might make sense.

Invariants

Invariants are rules that are imposed on data structures, objects, resource usage, and so on.

Example: linked lists must terminate with NULL.

Invariants tend to deal with the structure of data structures, objects, and other persistent resources like maybe hardware or network or database connections.

Why Contracts Matter

- **Working with others to define roles, boundaries, etc.**
- **Use contracts when programming solo to establish a good understanding of what you're about to do.**
- **Can be good documentation.**

There are lots of reasons to think about and write out your obligations for writing code.

If you're working with others, you'll likely use design by contract or some similar approach for breaking down the work and defining how the different bits of code work together.

Even if you're just working by yourself, thinking about functions as contracts will help you think through exactly what you're doing, about the full range of inputs, how that maps to outputs, and what could possibly go wrong.

Writing out code contracts can be a great starting point for documentation, and taking measures against burning the house down.

Episode 7

Linked Lists: Operations

In this episode, we're going to cover all the linked list operations from a high level. In the next two episodes, we're going to actually write some code that implements some of the homework. Between this and the next episodes, you should have enough to get the assignment done.

List Operations Revisited

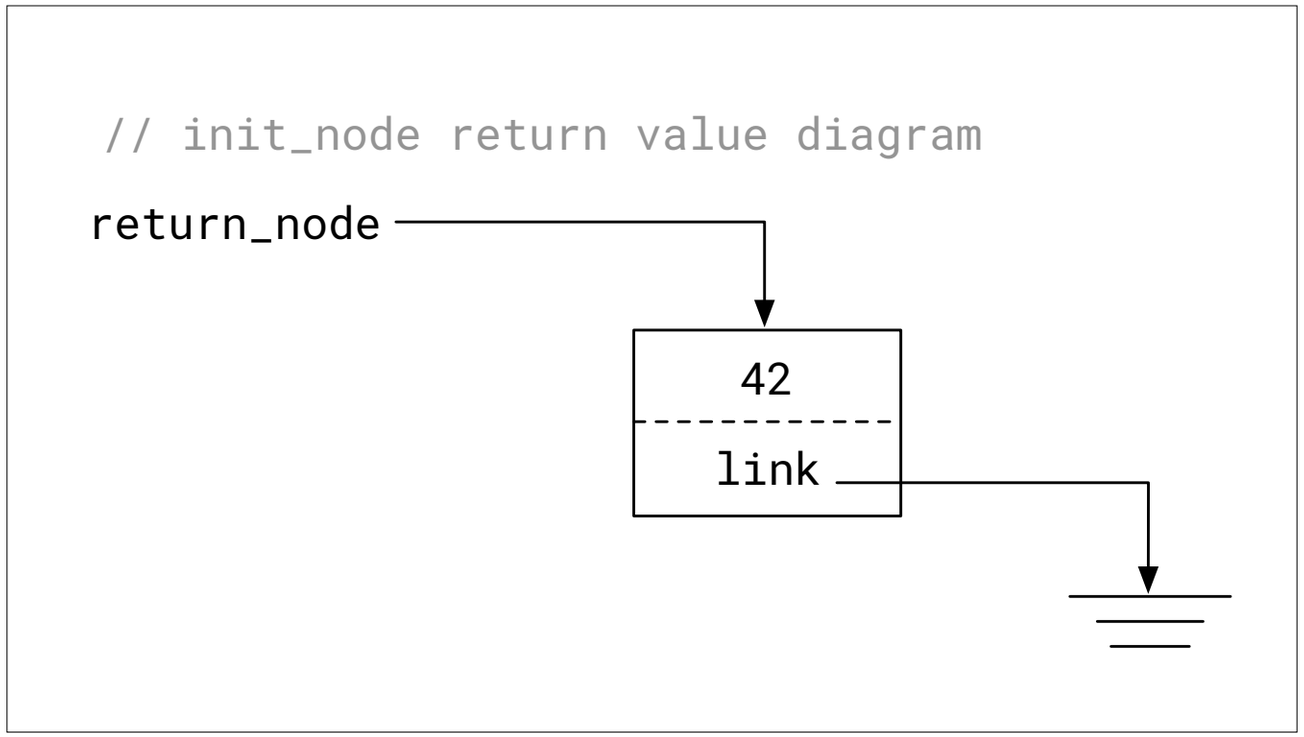
- initialize
- append
- insert
- remove
- contains
- size
- report

This is the list of operations that you'll implement for the homework. In the real world, there are other operations you'd likely want to do, but for our purposes, this list will suffice.

```
node* init_node(int data);
```

The first function to implement is `initialize`. It has instructions and a signature like this. Thinking about pre-conditions... do we want to have any limitations on what the input is? In this case, it doesn't matter at all. Any integer value is OK.

What about post-conditions? Well, the most obvious one is that we've initialized a node and returned a pointer to it. Thinking about the last episode about stack and heap memory, which do you think we're using here? The node has to be persistent after this `init_node` function exits, so... heap memory.



What other post-conditions are there? The instructions say the node points to NULL and holds the provided integer. So say we're initializing a new node with a value of 42, what we want is a diagram that looks like this.

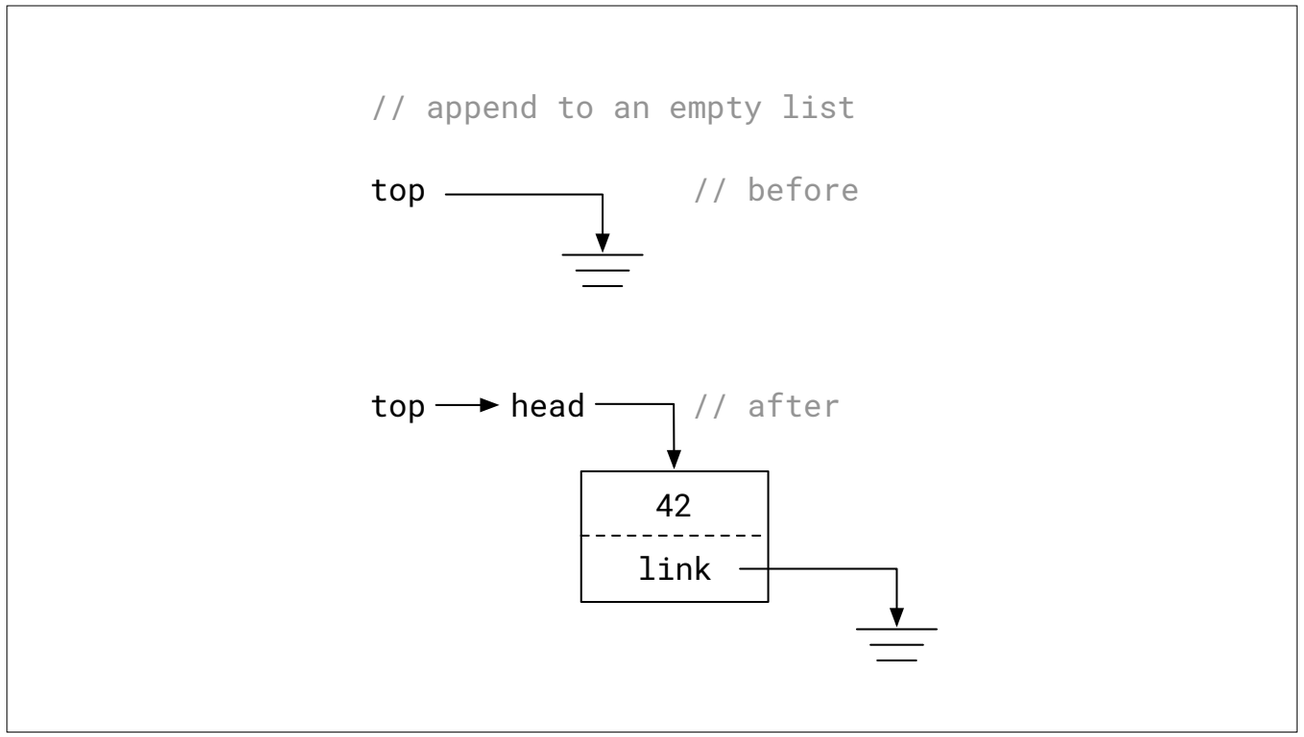
```
void append_data(node** top, int data);

void append(node** top, node* new_node);
```

Now we're getting into more interesting territory. The `append` function takes a pointer to an existing linked list and a pointer to a new node, and on exit the list has that node glommed on to the end of it. Here's the signature for both versions.

Wait a second, why are there two functions? And why don't they return anything?

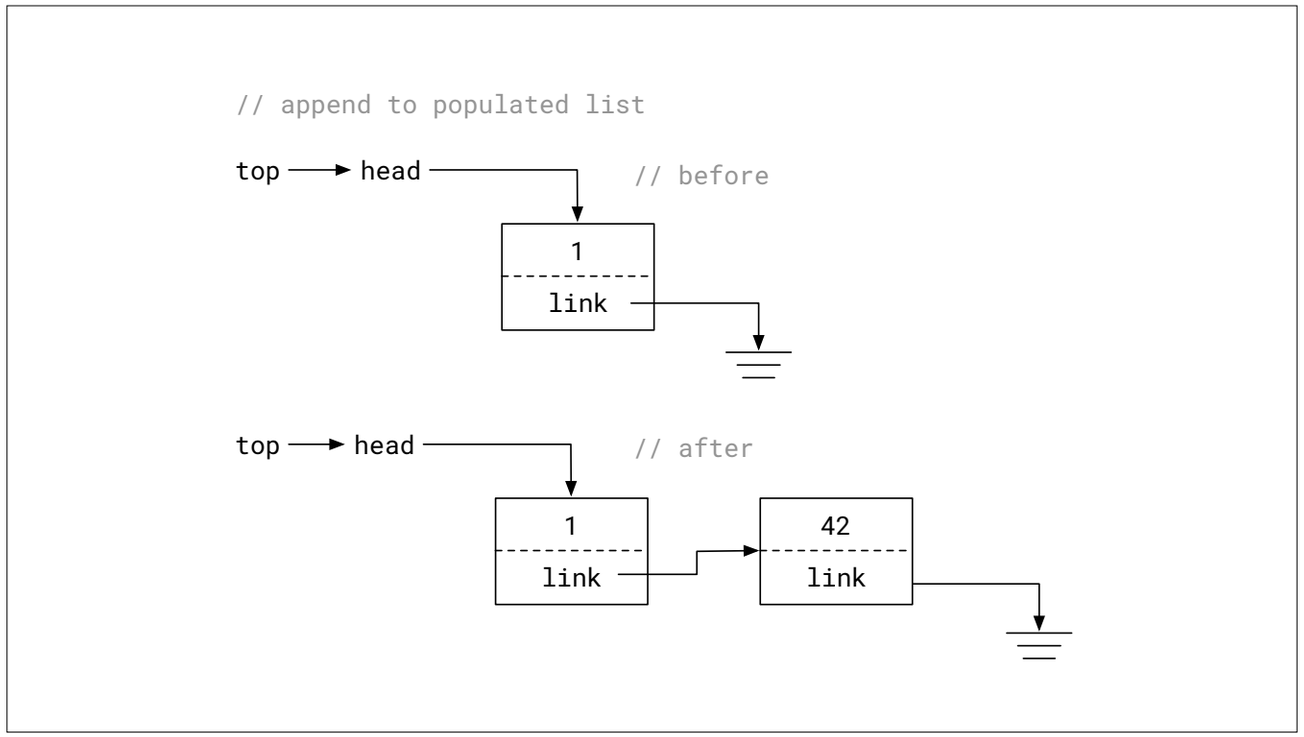
We actually don't *need* two functions. The `append_data` one is just a convenience function that initializes a new node and then uses the other `append` function to do the appending. So we'll just focus on the second function.

The `append` function doesn't return a value, but it does have a persistent change because it manipulates what the `top` variable points to. Think about what possible values we could give this function. What if we append to an empty list? Or what if the list has ten things in it? If we're appending to an empty list, that means the first node will be the `new_node` that we're passing in. But if the list isn't empty, we don't need to update the first node. That's why we pass in a node-star-star.



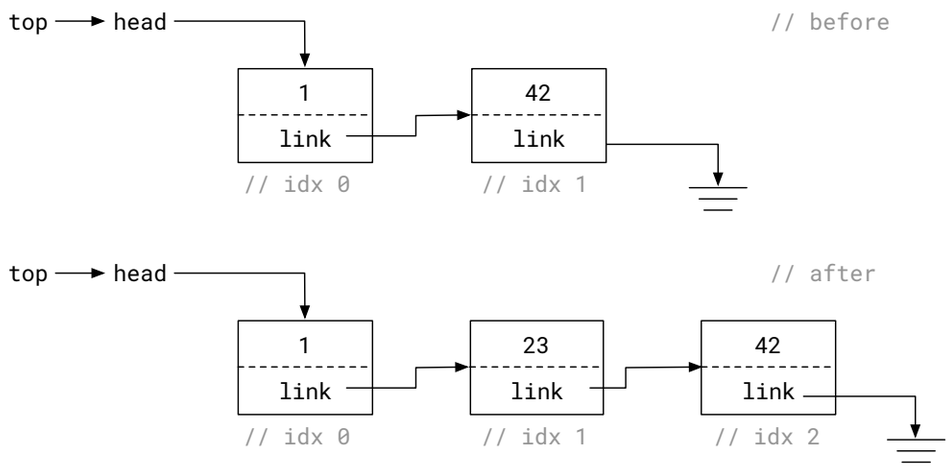
By the way, whenever we have a `node**` I'm going to call that `top` because it is a pointer to the first pointer. When the list is empty, `top` points to NULL, but when the list has stuff in it, `top` points to a valid pointer. I'm going to call that first pointer `head`. In Computer Science you'll see `head` and `tail` pretty often.

Anyway, about this drawing. We want to first recognize if and when we have the first situation, and if we do, we'll do whatever is needed to yield the second situation.



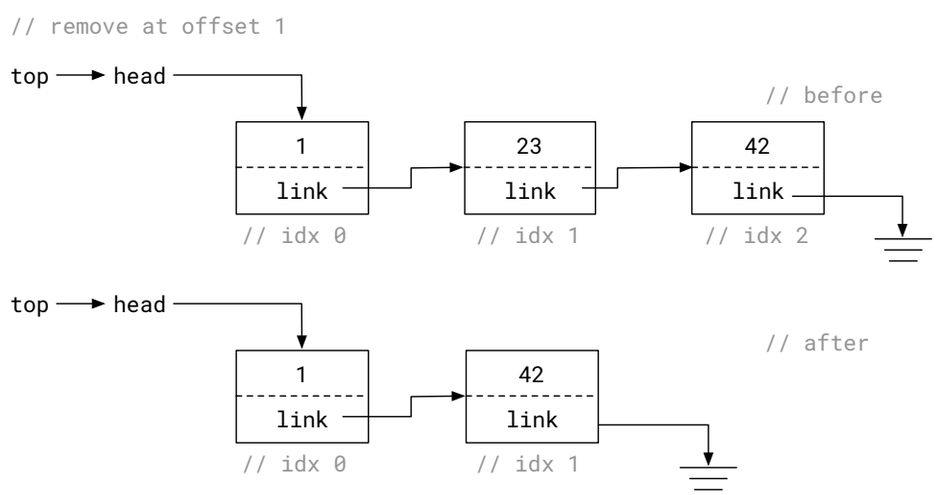
If the list isn't empty, it might have one thing in it, or a million. A one-item list looks like this. Notice that in this case, we don't have to modify `top` at all. We're only adjusting the node at the end of the list.

```
void insert(node** top, int offset, node* new_node);
// insert at offset 1 a node with value 23
```



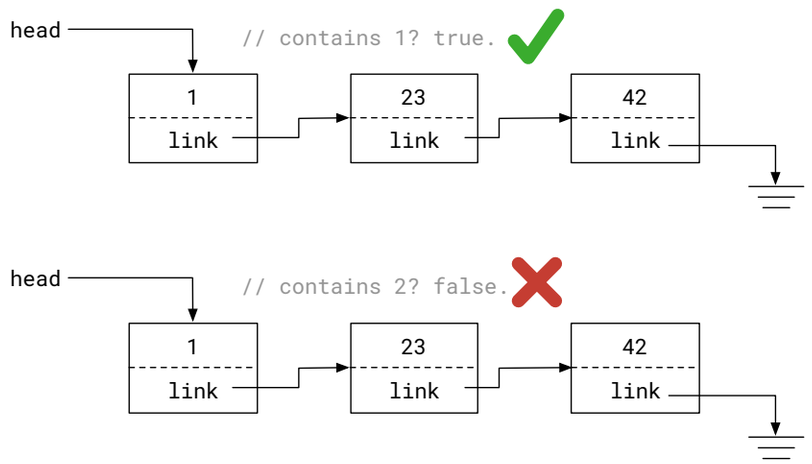
I'm going to quickly go over the remaining operations in diagram form, and it is up to you to identify the various cases, like how to handle empty lists. Insert gives a top pointer, an offset, and a new node to insert. The offset is the index the new node should have after it is in the list. So if we have an offset of zero, that means the new node will be the first on in the list. Notice the differences. We've updated pointers, both in a node that was in the list, and in the node that we've inserted.

```
void remove(node** top, int offset);
```



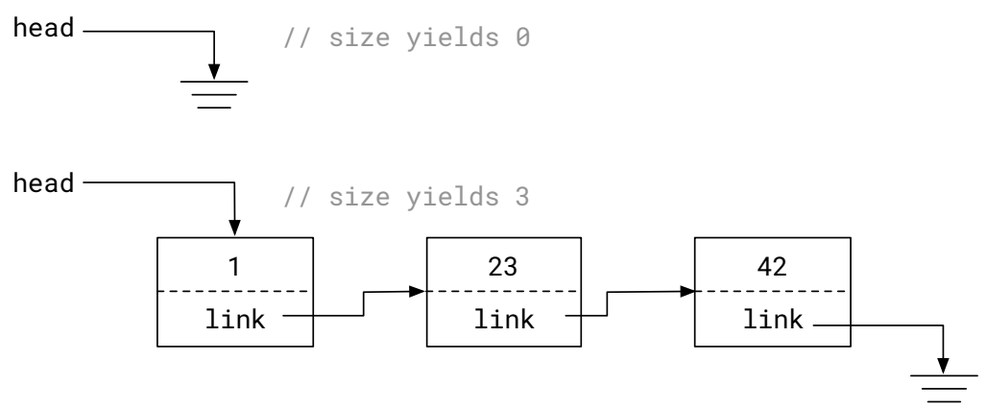
Remove will find the node at the given offset and remove it. There's three valid cases: First we might remove the very first node, at offset zero, which should change the head pointer. Second we might remove one of the nodes in the middle of the list somewhere, and that's what you're looking at here. Last, we might remove the very last node.

```
bool contains(node* head, int data);
```



Sometimes we're interested to know if a datastructure contains a particular value or reference. There are many forms of query functions, but in this homework you're just writing a `contains` routine that returns true or false depending on if the list has a node with a particular value. Notice that we're passing only a single pointer in to this function because there's no situation where it makes sense to modify the structure of the list itself.

```
int size(node* head);
```

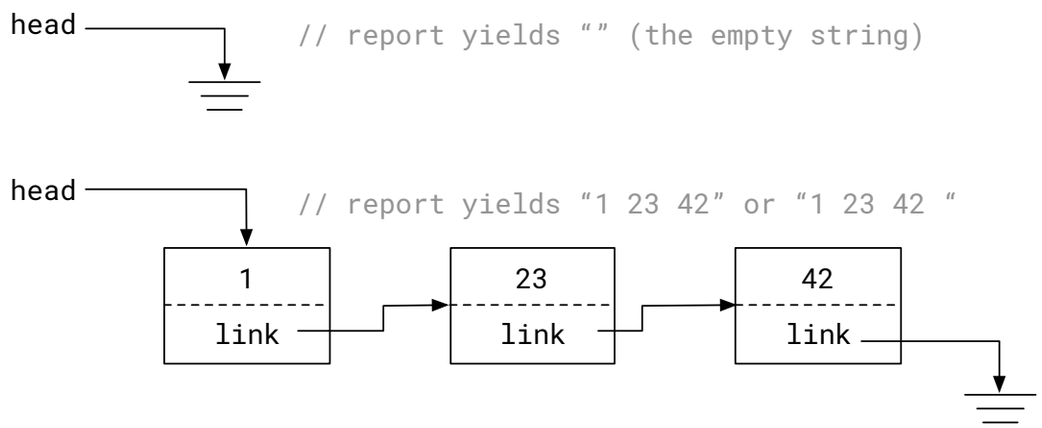


A common operation on data structures is to get its size. In different languages, libraries, or APIs this might be called something else, like length, len (spelled L-E-N), count, num, or whatever.

Our size function just counts the number of non-null references we can follow. If you're passed in a NULL pointer, the size is zero. Otherwise you can march through with a cursor and keep count of how many non-null nodes you've seen.

Notice we again get a single pointer here, for the same reason as the contains function.

```
std::string report(node* head);
```



The last function for the homework is a function that converts the datastructure to a string. This is often used for debugging, but has other uses too. If you're given a null pointer, just return an empty string. Otherwise, scroll through all the nodes and append the value and a space to a string, and at the end return it.

```cpp
#include <iostream> // gives cout
#include <string>   // gives string

using namespace std; // avoid writing std::string and std::cout

int main() {
  string message =  "this";
  message        += " message";
  message        =  message + " was built out of parts.";
  cout << message << endl;
}
```

```
this message was built out of parts.
```

This is an example of how to create and print out strings. Two things here. First, notice there are different ways to append to a string, either using the += operator, or by using the string variable plus whatever you want to attach to it. Second, notice that here we are just printing out the string, but in your homework you'll need to _return_ the string instead of printing it out.