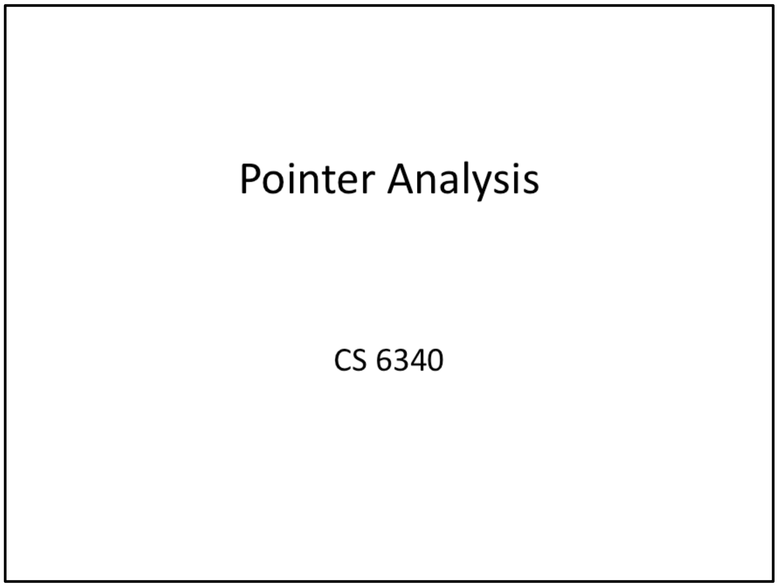


Pointer Analysis

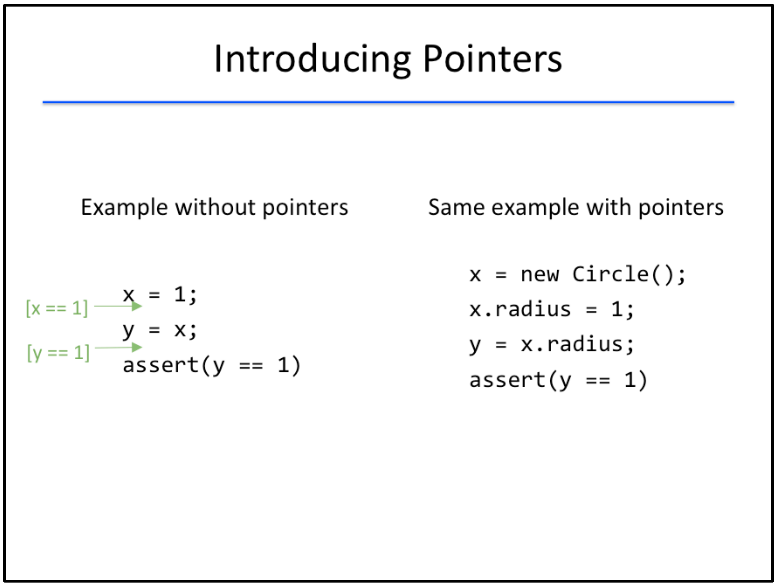
CS 6340

{HEADSHOT}

Previously, we learned how to reason about the flow of primitive data such as integers in a computer program. In this lesson, we will learn how to reason about the flow of non-primitive data, better known as pointers, objects, or references. This sort of analysis of a program is called a pointer analysis.

Pointers are prevalent in mainstream programming languages like C, C++, Java, and even Python. Therefore, pointer analysis is fundamental to any static analysis for reasoning about the flow of data in programs written today.

By the end of this lesson, you will have learned what pointer analysis is capable of doing and how you can incorporate it into your own dataflow analysis.



Introducing Pointers

Example without pointers

```
           x = 1;
[x == 1] →
           y = x;
[y == 1] →
           assert(y == 1)
```

Same example with pointers

```
x = new Circle();
x.radius = 1;
y = x.radius;
assert(y == 1)
```

Let's begin with a dataflow analysis for an example program without pointers: just two integer variables, x and y. Notice that this program can be expressed in the WHILE language that we introduced in the previous lesson, though we'll extend that language with the boolean operator == for the sake of clarity. Suppose the goal of our dataflow analysis is to prove the assertion that y equals 1 at the end of the program.

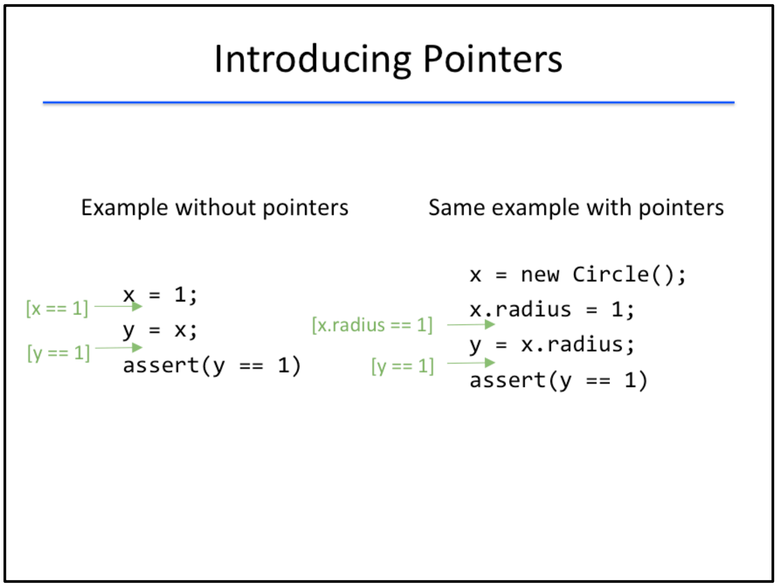
We can perform a forward, must analysis for this purpose. It begins by analyzing the assignment to x and infers that the value of x at this program point must be 1 ([x == 1] appears). It then analyzes the assignment to y, and infers that since the value of x before the assignment is 1, then the value of y after the assignment must also be 1 ([y==1] appears). The analysis thus proves that the assertion y==1 is valid at this point in the program.

Now let's slightly change this example to use pointers. Besides assignments, we see three new kinds of statements here, which we will need in order to meaningfully talk about pointer analysis in this lesson. Let's look at each of them in turn.

In the modified program, the first new type of statement is the object allocation statement, which uses the keyword "new". Much like in C++ and Java, this statement allocates memory for a new object of type Circle and then sets x to be the location of that allocated section of memory.

An object can have fields which we can read from and write to. In the statement "x.radius = 1", we access the memory allocated to the circle that x refers to, and then we access the portion of that allocated memory dedicated to the "radius" field of the circle object, and then we write the integer 1 to that space in memory. This operation is called a "field write".

By contrast, in the following statement "y = x.radius", we access the memory allocated to the circle that x refers to, then we read the integer in the section of the memory dedicated to the radius field of x, and then we copy that integer to another location in memory which stores the value of variable y. This operation is called a "field read".



Introducing Pointers

Example without pointers

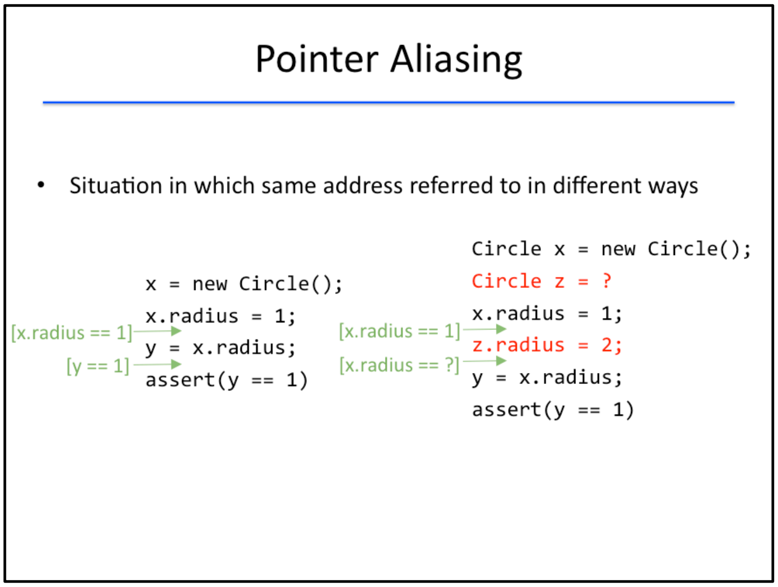
```
[x == 1]    x = 1;
            y = x;
[y == 1]    assert(y == 1)
```

Same example with pointers

```
                 x = new Circle();
[x.radius == 1]  x.radius = 1;
                 y = x.radius;
[y == 1]         assert(y == 1)
```

Now let's perform our analysis on this new program. We begin by analyzing the assignment to x.radius. We can infer that the value of x.radius at this program point must be 1 (write [x.radius == 1]). We then analyze this assignment to y, and we can infer that since the value of x.radius before the assignment is 1, the value of y after the assignment must also be 1 (write [y == 1]).
Our analysis therefore proves that this assertion is valid.

Notice that, this time, our analysis had to track the values of expressions more complex than variables, notably x.radius.



Pointer Aliasing

- Situation in which same address referred to in different ways

```
                 x = new Circle();
                 x.radius = 1;
[x.radius == 1] →
                 y = x.radius;
       [y == 1] →
                 assert(y == 1)
```

```
                 Circle x = new Circle();
                 Circle z = ?
                 x.radius = 1;
[x.radius == 1] →
                 z.radius = 2;
[x.radius == ?] →
                 y = x.radius;
                 assert(y == 1)
```

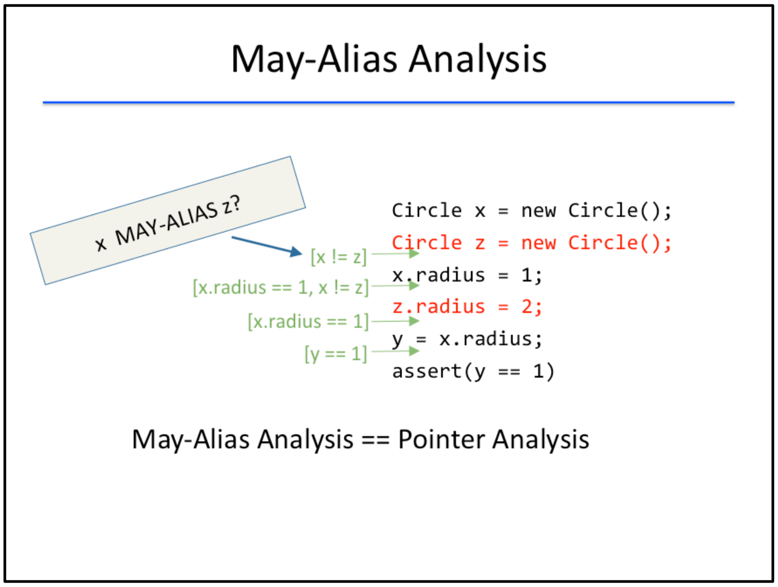
Expressions built using pointers, such as x.radius, allow the same memory address to be referred to in different ways. This situation is called pointer aliasing. The example we just looked at (gesture to example on left) did not have any pointer aliasing, since we had only one Circle pointer.

Let's look at a slightly different example that does have pointer aliasing and see what challenges it poses to our analysis (bring up example on right).

In this example, we have two Circle pointers, denoted x and z, but let's not commit yet to what z points to. Also note this additional assignment statement that writes 2 to the radius field of the Circle denoted by z.

Our analysis proceeds as before. After this assignment (point to statement x.radius = 1), we infer that the value of expression x.radius is 1. But after this assignment (point to statement z.radius = 2), our analysis is stuck: we do not know whether the value of expression x.radius should remain 1 or become 2. The answer depends on whether or not z is an alias of x.

Let's consider the two cases: one in which z denotes a different circle than x, and the other in which z denotes the same circle as x.



May-Alias Analysis

x MAY-ALIAS z?

```
Circle x = new Circle();
Circle z = new Circle();
x.radius = 1;
z.radius = 2;
y = x.radius;
assert(y == 1)
```

[x != z]

[x.radius == 1, x != z]

[x.radius == 1]

[y == 1]

May-Alias Analysis == Pointer Analysis

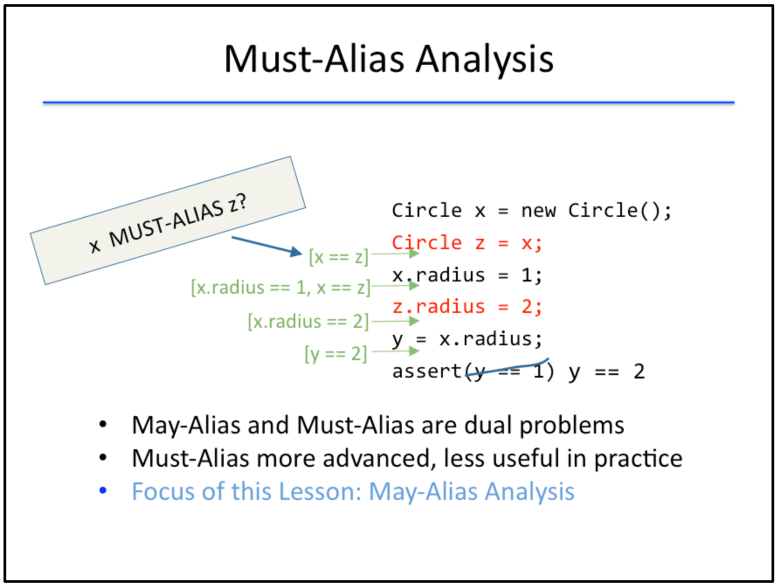
Here, let’s suppose x and z denote different Circles (underline new Circle()).

In this case, our analysis proceeds as follows.

After this assignment to z, we infer that x and z denote different circles. Continuing further, after this assignment to x.radius, we infer that the value of x.radius is 1. Now we analyze the assignment to z.radius. This time, we conclude that the value of x.radius remains 1 after this assignment because we tracked the fact x != z. Finally, we inspect this assignment (point to assignment) to y, and we infer that, since the value of x.radius is 1 before the assignment, the value of y must be 1 after the assignment, thereby proving the assertion.

To recap, our analysis was able to prove this assertion by tracking the fact that it is NOT true that x and z may alias (box and arrow appear). An analysis that is dedicated to proving facts of this form is called a MAY-alias analysis. MAY-alias analysis is also what we call pointer analysis.

At this point, you might be wondering: just as we had MAY vs. MUST dataflow analyses, is there a counterpart to MAY-alias analysis? The answer is yes, and, as you might expect, it is called MUST-alias analysis.



Must-Alias Analysis

x MUST-ALIAS z?

```
[x == z]
[x.radius == 1, x == z]
[x.radius == 2]
[y == 2]
```

```
Circle x = new Circle();
Circle z = x;
x.radius = 1;
z.radius = 2;
y = x.radius;
assert(~~y == 1~~) y == 2
```

- May-Alias and Must-Alias are dual problems
- Must-Alias more advanced, less useful in practice
- Focus of this Lesson: May-Alias Analysis

To understand must-alias analysis, let's consider the alternative case in our example program, where x and z denote the same circle (underline x). In other words, x and z are aliased.

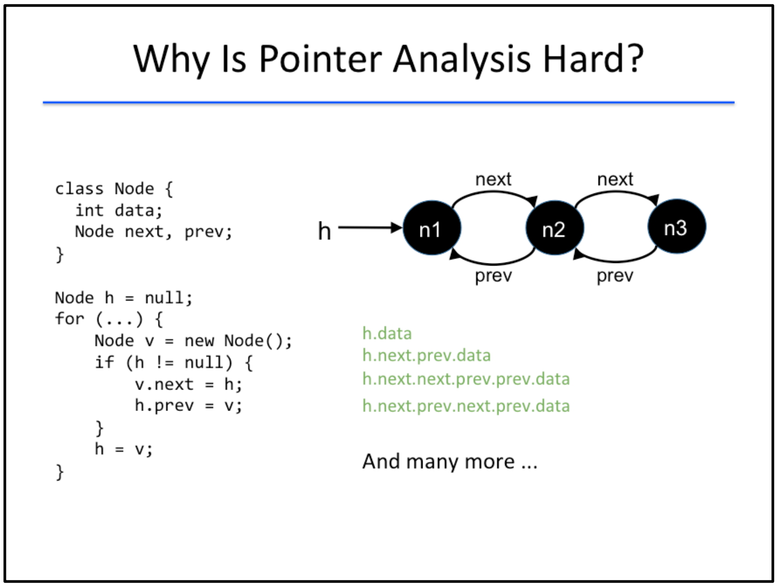
In this case, our analysis proceeds as follows.

After this assignment to z (point to statement), we infer that x and z denote the same circle. Continuing further, after this assignment to x.radius (point to statement), we infer that the value of x.radius is 1. And after analyzing the assignment to z.radius, we conclude that the value of x.radius becomes 2 after this assignment. We're able to conclude this fact because we tracked the fact that x and z must alias (box and arrow appear).

Finally, we look at the assignment to y. We infer that since the value of x.radius is 2 before the assignment, the value of y must be 2 after the assignment. The analysis thus fails to prove the assertion. Indeed, this assertion is incorrect. The correct assertion should be y == 2, which is what our analysis also proves (strike out y == 1 and hand-write y==2).

To recap, our analysis was able to prove this assertion by tracking the fact that x and z MUST alias.

May-alias analysis and must-alias analysis are duals of each other. But the technical machinery needed for must-alias analysis is far more advanced than that for may-alias analysis. Also, may-alias analysis is useful for many more practical dataflow analysis problems than must-alias analysis. Therefore, in this lesson, we will focus on may-alias analysis, which is also called pointer analysis.



Why Is Pointer Analysis Hard?

```
class Node {
  int data;
  Node next, prev;
}

Node h = null;
for (...) {
    Node v = new Node();
    if (h != null) {
        v.next = h;
        h.prev = v;
    }
    h = v;
}
```

h.data
h.next.prev.data
h.next.next.prev.prev.data
h.next.prev.next.prev.data

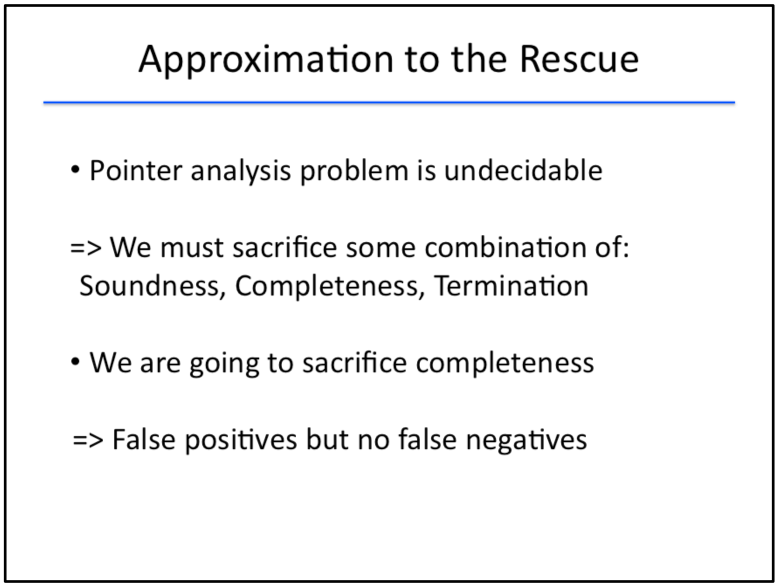
And many more ...

Before we dive into how to do pointer analysis, it is worthwhile to look at why we need a separate type of analysis for pointers. This will help make the motivation for the pointer analysis algorithm more clear.

Dataflow analysis in the presence of pointers is more challenging than dataflow analysis in the absence of pointers for a couple of reasons. Let's take a look at the following data structure, a doubly linked list, created by this example program (code snippet and graph appear). Each vertex is a Node object in memory, and each Node object has a data field and two pointer fields, next and prev, capable of pointing to other Nodes. A precise dataflow analysis of this program would need to keep track of all possible ways of accessing each Node's data.

For example, the data field of the Node denoted n1 could be referred to in many different ways. One way to refer to it is simply h.data. Another way to refer to it is h.next.prev.data. Yet more ways to refer to it are h.next.next.prev.prev.data, h.next.prev.next.prev.data, and so on.

Tracking all these different expressions is inefficient at best and infeasible at worst: in the presence of cycles, like in this example, there are infinitely many ways of referring to the same piece of data.



Approximation to the Rescue

- Pointer analysis problem is undecidable

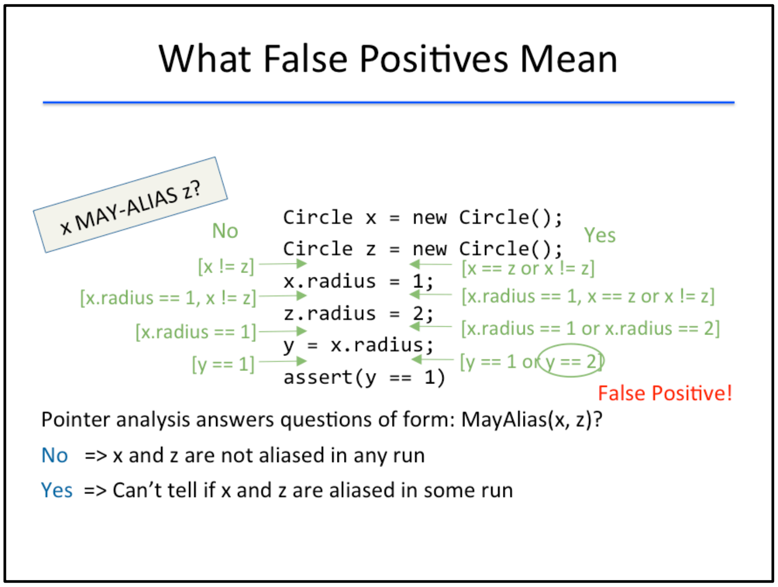
=> We must sacrifice some combination of:
Soundness, Completeness, Termination

- We are going to sacrifice completeness

=> False positives but no false negatives

As was the case for dataflow analysis in the absence of pointers, the problem of deciding whether two pointers alias is in general an undecidable problem. In other words, there is no algorithm that always terminates and perfectly decides whether two pointers alias.

The solution to this problem of undecidability is to sacrifice completeness, just as we did for dataflow analysis in the absence of pointers. This means, in exchange for the possibility of obtaining false positives, we can design an alias-detection algorithm that terminates and never gives false negatives.



What False Positives Mean

x MAY-ALIAS z?

No

[x != z]

[x.radius == 1, x != z]

[x.radius == 1]

[y == 1]

```
Circle x = new Circle();
Circle z = new Circle();
x.radius = 1;
z.radius = 2;
y = x.radius;
assert(y == 1)
```

Yes

[x == z or x != z]

[x.radius == 1, x == z or x != z]

[x.radius == 1 or x.radius == 2]

[y == 1 or y == 2]

False Positive!

Pointer analysis answers questions of form: MayAlias(x, z)?

No => x and z are not aliased in any run

Yes => Can’t tell if x and z are aliased in some run

It is worth being a bit more precise in what we mean by the term “false positive.” Let’s revisit our earlier example.

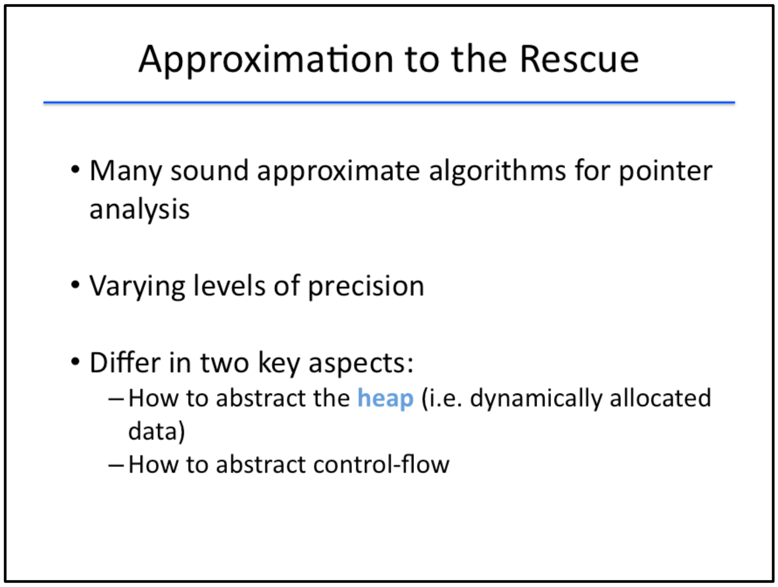
Remember that the question we are asking in this problem is: “Is it possible for two given pointers to be aliases of one another in some execution of this program?”

A shorthand version of this question is the boolean function “x MayAlias z”. This function returns NO if there is no possibility that x and z are aliases, as is the case in this example.

Take a moment to convince yourself that this answer enables our dataflow analysis to eventually prove the assertion at the end of this program (bring up the green steps on the left).

Conversely, x MayAlias z returns YES if we cannot determine whether x and z are aliases or not. In other words, YES does not mean x and z MUST be aliases: it just means they may or may not be aliases. If x MayAlias z returns YES when x and z are not actually aliases, we consider this a “false positive.”

Let’s take a look at how a false positive manifests in this example. Suppose x MayAlias z returns Yes. Then, the most accurate information that our dataflow analysis can safely infer at this point is that x may or may not be equal to z. Continuing this reasoning, our dataflow analysis concludes at the end of the program that the value of y may be either 1 or 2. The analysis thus fails to prove the assertion that the value of y must always be 1 at this point. The conclusion that the value of y may be 2 is a false positive whose existence can be traced back to the pointer analysis answering Yes to the question “x MayAlias z” when in fact x and z are not aliases.



Approximation to the Rescue

- Many sound approximate algorithms for pointer analysis
- Varying levels of precision
- Differ in two key aspects:
  - How to abstract the **heap** (i.e. dynamically allocated data)
  - How to abstract control-flow

There are many sound but approximate algorithms to the problem of pointer analysis.

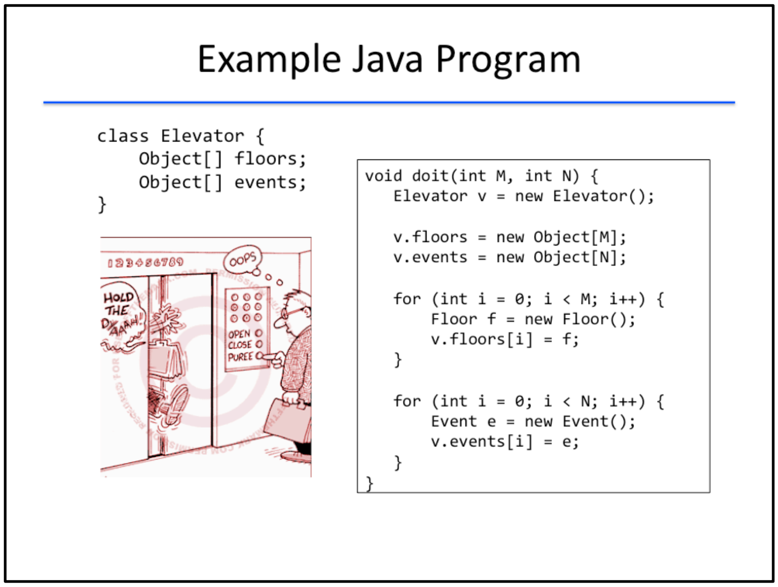
All these approximate algorithms generate false positives in certain circumstances, but they differ in their precision; that is, their false-positive rate.

The approximations that these algorithms perform differ in two key aspects: how they abstract program data, in particular dynamically allocated data, which we will call the heap; and how they abstract control-flow.

We already saw an abstraction of control-flow in the previous lesson on dataflow analysis where we approximated all branch conditions in the program's control-flow graph using non-deterministic choice.

Pointer analyses typically go further in that they ignore control flow entirely and instead look at the program as a set of unordered statements.

Let's dive deeper into data and control-flow abstractions for pointer analyses using an example program.



Example Java Program

```
class Elevator {
    Object[] floors;
    Object[] events;
}
```

123456789
OOPS
HOLD THE DAAAH!
OPEN
CLOSE
PUREE

```
void doit(int M, int N) {
   Elevator v = new Elevator();

   v.floors = new Object[M];
   v.events = new Object[N];

   for (int i = 0; i < M; i++) {
       Floor f = new Floor();
       v.floors[i] = f;
   }

   for (int i = 0; i < N; i++) {
       Event e = new Event();
       v.events[i] = e;
   }
}
```

Throughout this lesson, we will use this Java program to illustrate the key concepts of pointer analysis. This program constructs a representation of an elevator.

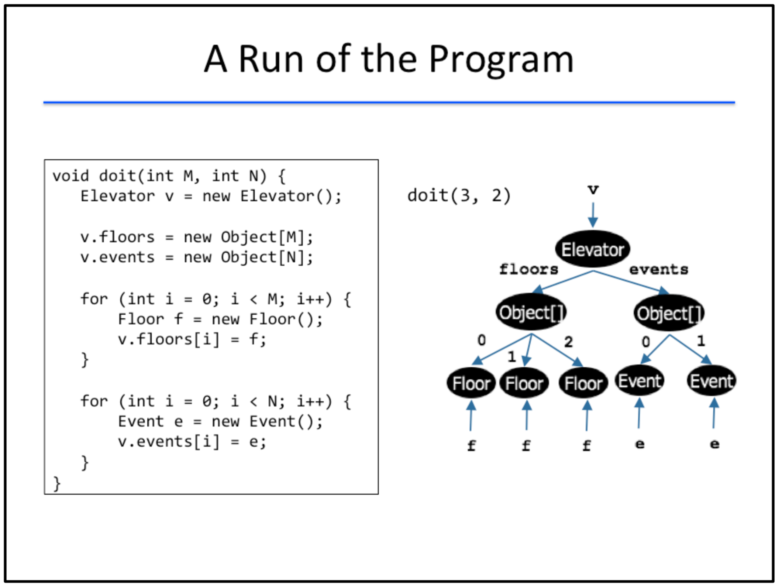
An Elevator object has two fields.

One field is an array of Floor objects (point to floors field) representing different floors in a building, such as the basement, 1st floor, 2nd floor, and so on.

The other field is an array of Event objects (point to events field). An example event is a person pushing a button for the 2nd floor in an elevator currently on the 5th floor.

The doit function takes as input the number of floors M and the number of events N. It starts out by creating an object of class Elevator. It then initializes the floors and events fields of the created Elevator object.

Let’s take a look at a sample concrete run of this program.



A Run of the Program

```
void doit(int M, int N) {
   Elevator v = new Elevator();

   v.floors = new Object[M];
   v.events = new Object[N];

   for (int i = 0; i < M; i++) {
       Floor f = new Floor();
       v.floors[i] = f;
   }

   for (int i = 0; i < N; i++) {
       Event e = new Event();
       v.events[i] = e;
   }
}
```

```
doit(3, 2)
```

In this run, we will create an elevator for a building with three floors and two events. That is, we will assume that the doit function is called with M == 3 and N == 2.

When the doit function is called, the allocation on the right-hand side of this line (point to allocation) is evaluated first. This allocation calls the Elevator's constructor method, which we do not show. Having completed the construction of the Elevator object, we assign the address of this Elevator object in memory to the variable v (draw arrow pointing to Elevator node and label it v).

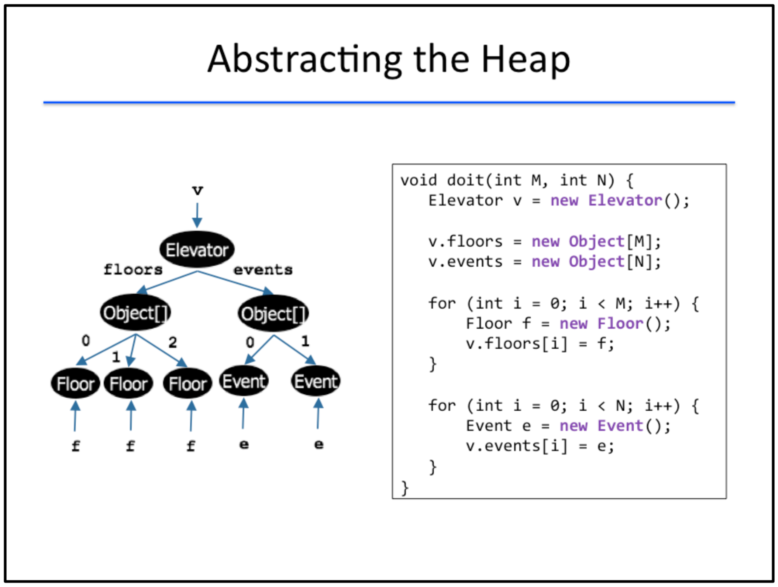
In the next line of the program, we first allocate an Object array with 3 cells, and then we write the location of this array in memory to the floors field of the Elevator object we're constructing (draw arrow from Elevator to left-hand Object[] node, label arrow by "floors").

In the next line, the same procedure happens, except now we're allocating an Object array with just 2 cells, and we're writing the location of this array to the events field of the Elevator object (draw arrow from Elevator to right-hand Object[] node, label arrow by "events").

In the 0th iteration of the first for-loop (point at first for-loop), we take four steps. We allocate a new Floor object, assign the Floor object's memory address to the pointer f (draw vertical arrow under left-most Floor node, label arrow by f), read the floors field of the Elevator object being constructed, and do a field-write of the address in f to the 0th cell of the Object array at the address contained in the floors field (draw arrow from leftmost Object[] node to leftmost Floor node, label arrow by 0). We then repeat this procedure for the second and third iterations of this loop (draw remaining arrows for Floor objects and label them appropriately).

Similarly, for the 0th iteration of the second for-loop (point at second for-loop), we allocate a new Event object, assign the Event object's memory address to the pointer e (draw vertical arrow under left-most Event node, label arrow by e), read the events field of the Elevator object, and do a field-write of the address in e to the 0th cell of the Object array at the address contained in the events field (draw arrow from rightmost Object[] node to leftmost Event node, label arrow by 0). We then repeat this procedure for the second iteration of this loop (draw remaining arrows for Event object and label them appropriately).

This concludes the operation of the doit function.



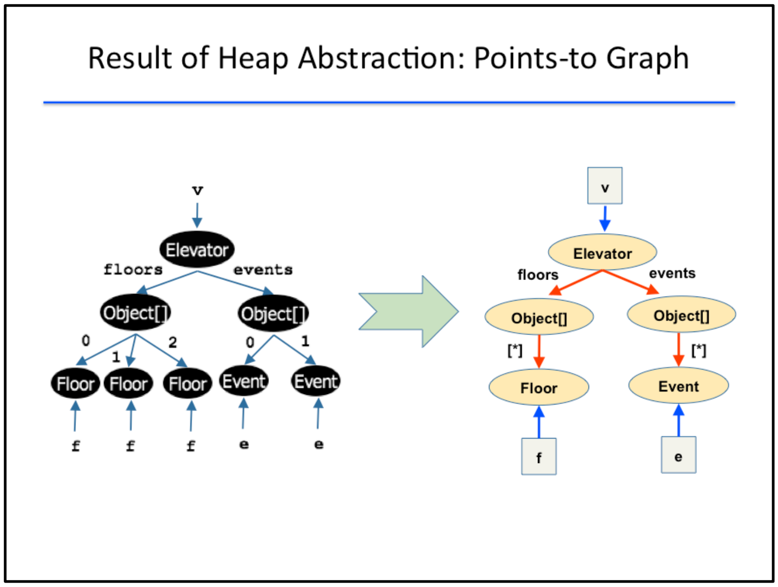
Abstracting the Heap

```
void doit(int M, int N) {
   Elevator v = new Elevator();

   v.floors = new Object[M];
   v.events = new Object[N];

   for (int i = 0; i < M; i++) {
       Floor f = new Floor();
       v.floors[i] = f;
   }

   for (int i = 0; i < N; i++) {
       Event e = new Event();
       v.events[i] = e;
   }
}
```

This run of the elevator program created only three floors and two events, but a pointer analysis must be able to reason about each run with M floors and N events, for any value of M and N.

Pointer analysis achieves this by abstracting the heap. There are many possible schemes to abstract the heap, each of which strikes a different tradeoff between precision and efficiency. One of these schemes abstracts objects based on the site at which they are allocated in the program. We will look at other schemes later in this lesson.

The elevator program has five allocation sites. Therefore, pointer analysis operates on the following graph, which conflates all objects allocated at the same site into a single node of the graph (replace code snippet by new graph).

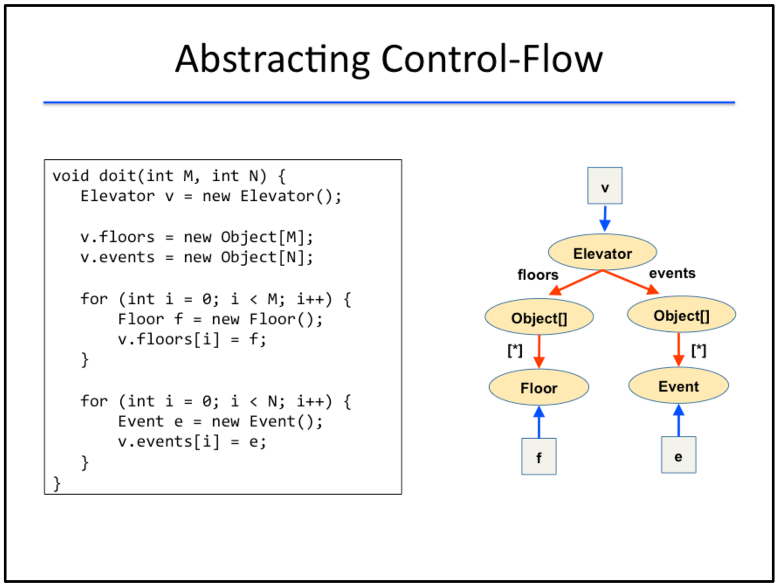


Result of Heap Abstraction: Points-to Graph

In particular, it collapses all the Floor nodes into a single Floor node, and all the Event nodes into a single Event node. Instead of labeling the pointers from the Object array nodes to these individual objects by array indices, we use the nondeterministic-choice symbol asterisk. And also observe that the variables f and e now point to a single node each instead of multiple nodes.

The type of graph that this heap abstraction produces is called a "points-to" graph. In a general points-to graph, there are two kinds of nodes: variables and allocation sites. Variables will be denoted by boxes, and allocation sites will be represented by ovals.

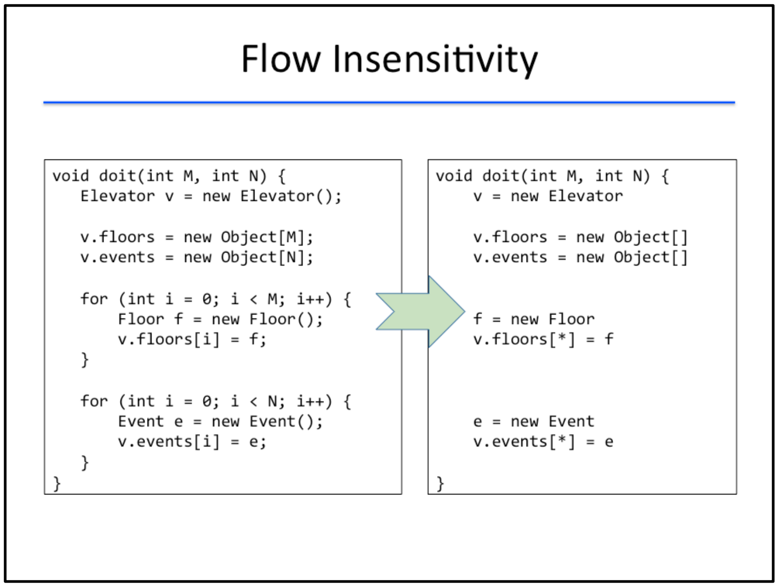
Since the points-to graph relation is an asymmetric relationship (that is, pointers between data need not go both ways), this is a directed graph. So we will represent edges with arrows. There are two types of edges in this graph. Arrows from a variable node to an allocation site node will be colored blue, and arrows from one allocation site node to another allocation site node will be colored red and labeled by a field name.



Abstracting Control-Flow

```
void doit(int M, int N) {
   Elevator v = new Elevator();

   v.floors = new Object[M];
   v.events = new Object[N];

   for (int i = 0; i < M; i++) {
       Floor f = new Floor();
       v.floors[i] = f;
   }

   for (int i = 0; i < N; i++) {
       Event e = new Event();
       v.events[i] = e;
   }
}
```

Although points-to graphs are finite, it is too expensive in practice to track a separate such graph at each program point. This is in contrast to the dataflow analyses we learnt in the last lesson where we tracked a separate set of dataflow facts at each program point. Instead, most pointer analyses only track a single, global points-to graph for the entire program. They achieve this by abstracting control-flow (code on right-hand side fades in, shown on next slide).



Flow Insensitivity

```
void doit(int M, int N) {
   Elevator v = new Elevator();

   v.floors = new Object[M];
   v.events = new Object[N];

   for (int i = 0; i < M; i++) {
       Floor f = new Floor();
       v.floors[i] = f;
   }

   for (int i = 0; i < N; i++) {
       Event e = new Event();
       v.events[i] = e;
   }
}
```

```
void doit(int M, int N) {
    v = new Elevator

    v.floors = new Object[]
    v.events = new Object[]


    f = new Floor
    v.floors[*] = f




    e = new Event
    v.events[*] = e

}
```

The particular control-flow abstraction that is commonly used by pointer analyses is called flow insensitivity. Applying this abstraction to our example elevator program produces the following abstract program.

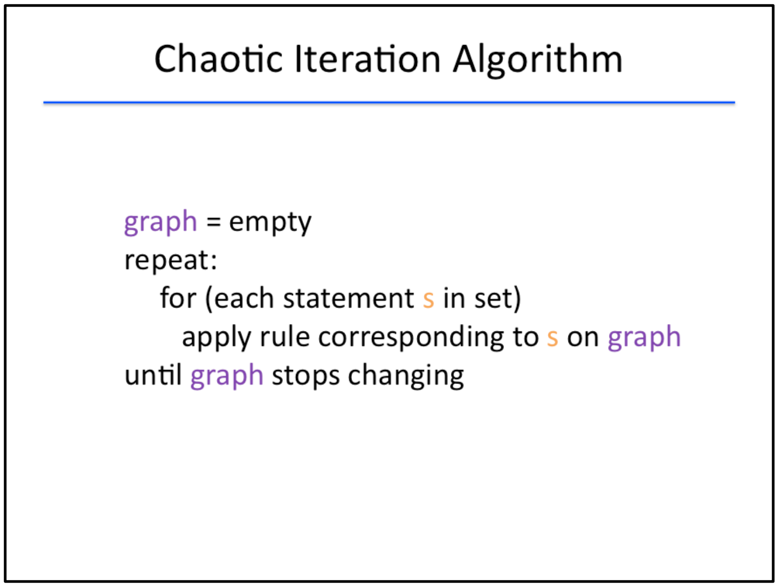
There are three major differences I'd like to highlight here:

- Notice that all control-flow features have been removed, including constructs like for-loops and also semicolons indicating sequentially ordered statements.
- All statements that do not affect pointers have also been removed. For example, the approximated code no longer has statements setting the integer i equal to 0 and incrementing i.
- Finally, array indices are replaced by nondeterministic choice, denoted by the asterisk symbol. This is similar to how conditions at branch points in dataflow analysis were replaced by nondeterministic choice.

This abstraction can be thought of as turning the program into an unordered set of statements. Even though we've written the statements in the rough order they appear in the original program, it is better to think about the statements as having no precedence over each other.

Even though this abstraction appears to lose a lot of information, we will see later in this lesson that it still able to prove interesting properties, such as the property that variables e and f do not alias.

Next, let's see how a pointer analysis algorithm builds a points-to graph for an arbitrary program.



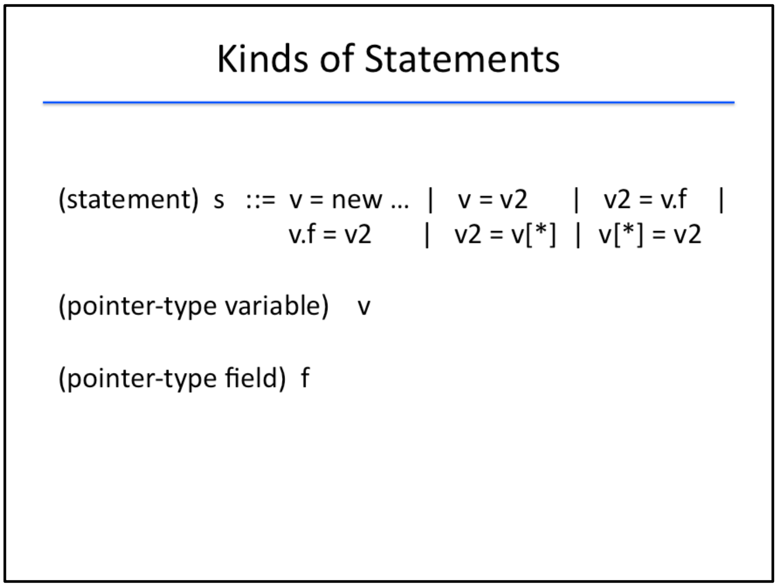
Chaotic Iteration Algorithm

```
graph = empty
repeat:
    for (each statement s in set)
        apply rule corresponding to s on graph
until graph stops changing
```

The pointer analysis algorithm follows a similar pattern as the dataflow analyses we saw earlier, so we again call the algorithm a Chaotic Iteration Algorithm.

We begin by starting with an empty graph. We take the set of statements obtained earlier by applying the flow-insensitivity approximation. And we iterate through each statement in this set, applying the rule corresponding to that statement to the graph we're building. We continue applying these rules until we iterate through the entire set of statements without making any changes to the graph. At that point, we terminate the algorithm.

Now, this is a basic algorithm for pointer analysis, and it sweeps many implementation details under the rug. in practice, pointer analysis algorithms will use a more efficient traversal of statements. They will also employ data structures that make it efficient to update the graph.



Kinds of Statements

```
(statement)  s   ::=  v = new …  |   v = v2      |   v2 = v.f    |
                      v.f = v2      |   v2 = v[*]  |  v[*] = v2
```

(pointer-type variable)   v

(pointer-type field) f

For pointer analysis, it suffices to consider the following kinds of statements. (These statements are similar to those you might find in a Java program.)

(point to each statement in turn)

The first is an object allocation statement, which assigns the address of a newly allocated object in memory to a pointer-type variable v.

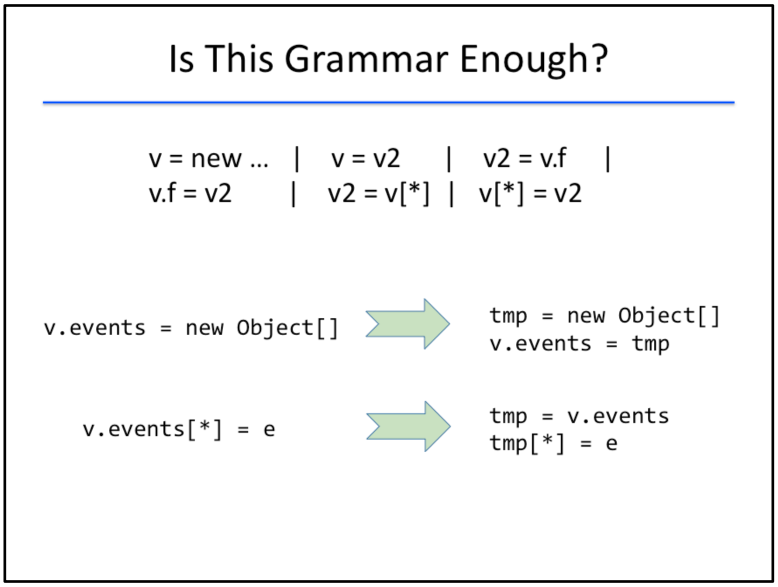
The second is an object copy statement, which copies the contents of one pointer-type variable to another. (In other words, we are writing the memory address stored in v2 to v.)

The third is a field-read statement, which reads the contents of a pointer-type field f from some object (here referenced by the pointer-type variable v) and stores those contents to another pointer-type variable v2.

The fourth is a field-write statement, which writes the contents of a pointer-type variable v2 to a pointer-type field f of an object.

Additionally, we can do a different type of field-read by reading the contents of some cell of an array of pointers and store those contents to another pointer.

Likewise, we can do a different type of field-write by reading the contents of some pointer-type variable and writing those contents to some cell in an array of pointers.



Is This Grammar Enough?

```
v = new …   |   v = v2      |   v2 = v.f    |
v.f = v2      |   v2 = v[*]  |  v[*] = v2
```

```
v.events = new Object[]   →   tmp = new Object[]
                              v.events = tmp

   v.events[*] = e        →   tmp = v.events
                              tmp[*] = e
```

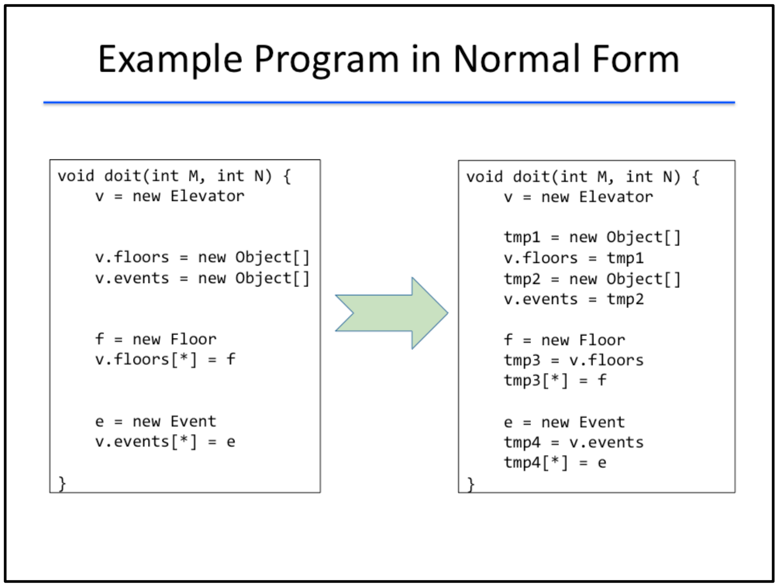
At this point, you might be wondering whether the grammar we just presented is sufficient to represent all the operations that we might need to reason about in order to determine whether two pointers may alias. The answer is yes.

We will not give a formal proof of this fact in this lecture. We will, however, show some examples of how to break down more complicated statements into compositions of these six simpler types of statements.

First, let's consider the statement "v.events = new Object[]" from our elevator program. This statement starts with an allocation of the Object array, and then it assigns the address of the new allocation to the field events of the Elevator object.

We can break this statement down into two separate operations. First, we allocate the new Object array and assign its address to a pointer-type variable, which we'll call "tmp." (This statement is of the form of the first of our six simple operations.) Then we'll assign the address stored in tmp to the events field of the Elevator object. (This statement is a field-write, our fourth kind of simple operation.)

Here's a second example statement: "this.events[*] = e". It performs a read of the events field of the Elevator object, and then it assigns the content of e to some cell in the Object array that v.events points to. We can again break this statement up into two separate statements of the forms above. First we have a field-read operation, assigning the content of v.events to the pointer variable tmp. So tmp now points to an array of pointers to Objects. Then we assign the content of e to a nondeterministically chosen element of this array. We've now broken the compound statement into two simple statements of the form specified in our grammar.



Example Program in Normal Form

```
void doit(int M, int N) {
    v = new Elevator


    v.floors = new Object[]
    v.events = new Object[]


    f = new Floor
    v.floors[*] = f


    e = new Event
    v.events[*] = e

}
```

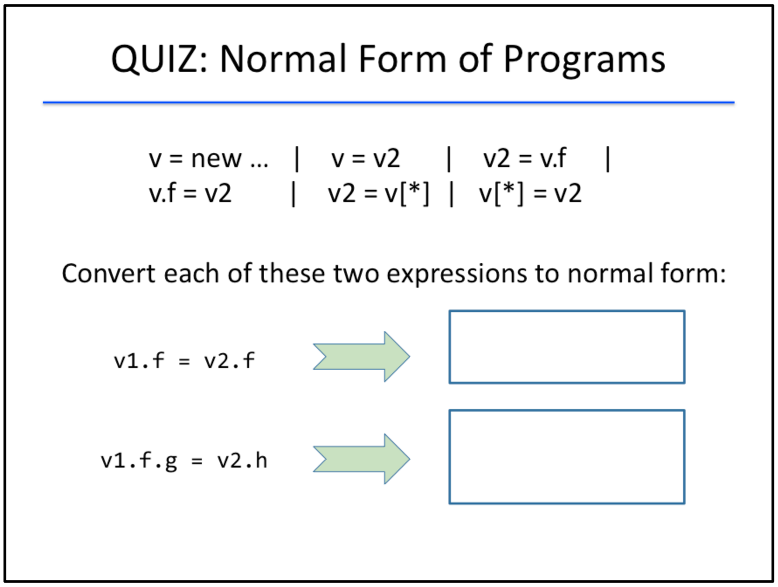
```
void doit(int M, int N) {
    v = new Elevator

    tmp1 = new Object[]
    v.floors = tmp1
    tmp2 = new Object[]
    v.events = tmp2

    f = new Floor
    tmp3 = v.floors
    tmp3[*] = f

    e = new Event
    tmp4 = v.events
    tmp4[*] = e
}
```

Here’s what happens if we apply these decomposition rules to all of the statements in our Elevator program.

Like we saw in the first decomposition example, these two statements are decomposed into an allocation and a pointer-based field-write.

The statements f = new Floor and e = new Event are already in forms specified by the grammar, so we don’t modify them.

However, we modify the assignments to the array cells of v.floors and v.events using the pattern we saw in the second example previously.



QUIZ: Normal Form of Programs

v = new … | v = v2 | v2 = v.f |
v.f = v2 | v2 = v[*] | v[*] = v2

Convert each of these two expressions to normal form:

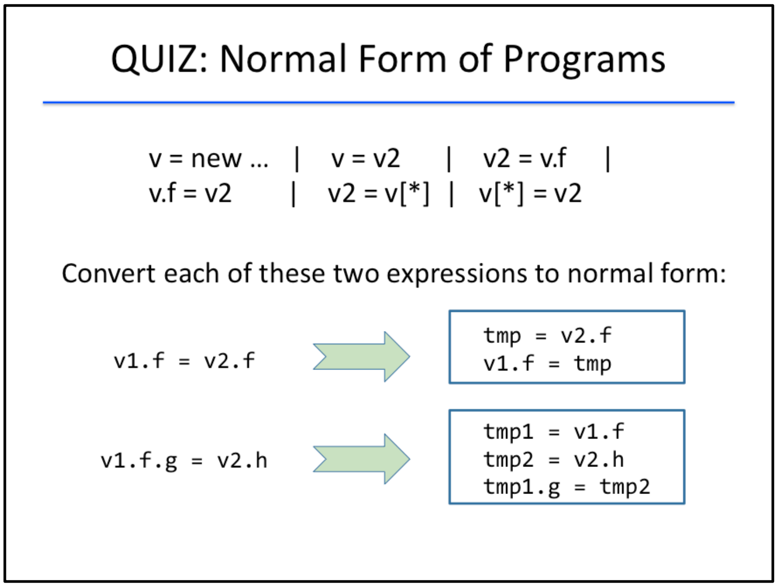
```
v1.f = v2.f
```

```
v1.f.g = v2.h
```

Let’s do a quick quiz to check your understanding of the form of programs that we’ll be using for pointer analysis, which we will call a normal form.

For each of these two statements, “v1.f = v2.f” and “v1.f.g = v2.h” fill in the box on the right with equivalent statements in the normal form specified by this grammar.

Note that you may need more than two simple statements to capture the entire compound statement.



QUIZ: Normal Form of Programs

v = new … | v = v2 | v2 = v.f |
v.f = v2 | v2 = v[*] | v[*] = v2

Convert each of these two expressions to normal form:

```
v1.f = v2.f
```

```
tmp = v2.f
v1.f = tmp
```

```
v1.f.g = v2.h
```

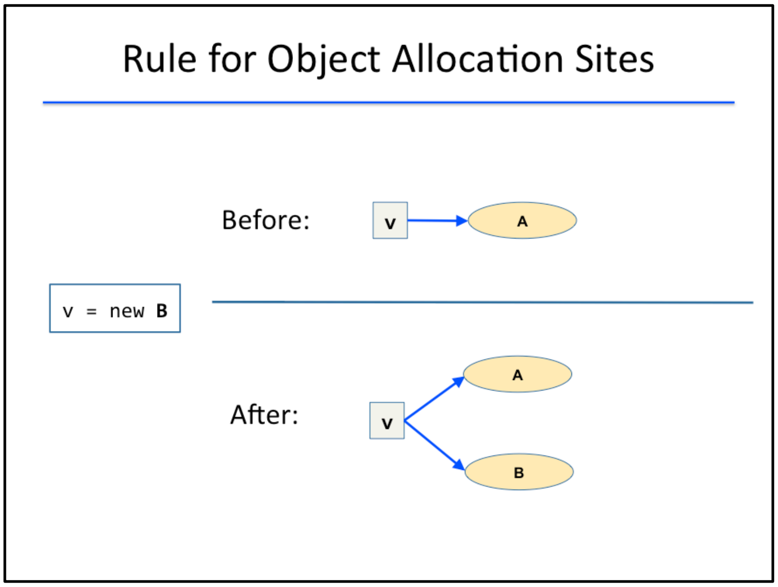
```
tmp1 = v1.f
tmp2 = v2.h
tmp1.g = tmp2
```

{SOLUTION SLIDE}

Let's look at some example solutions. Your answers might not look exactly the same, but they should have the same patterns.

For the first statement, v1.f = v2.f, we first read the f field of v2, and then we write the contents of v2.f to the field f of v1. Therefore, we need both a field-read statement and a field-write statement to capture this entire operation. I'll first assign the contents of v2.f to tmp (tmp = v2.f appears), and then I'll assign the contents of tmp to v1.f (v1.f = tmp appears).

For the second statement, v1.f.g = v2.h, we'll need more than two simple statements. We need to access the contents of v2.h, so that will be the field-read operation tmp2 = v2.h (tmp2 = v2.h appears). We also need to access the contents of v1.f in order to reach g, so that's another field-read operation: tmp1 = v1.f (tmp1 = v1.f appears). Finally, we need to write the contents of v2.h (in tmp2) to the field g of v1.f (in tmp1). We use a field-write operation to do so: tmp1.g = tmp2 (tmp1.g = tmp2 appears).



Rule for Object Allocation Sites

`v = new B`

Before:

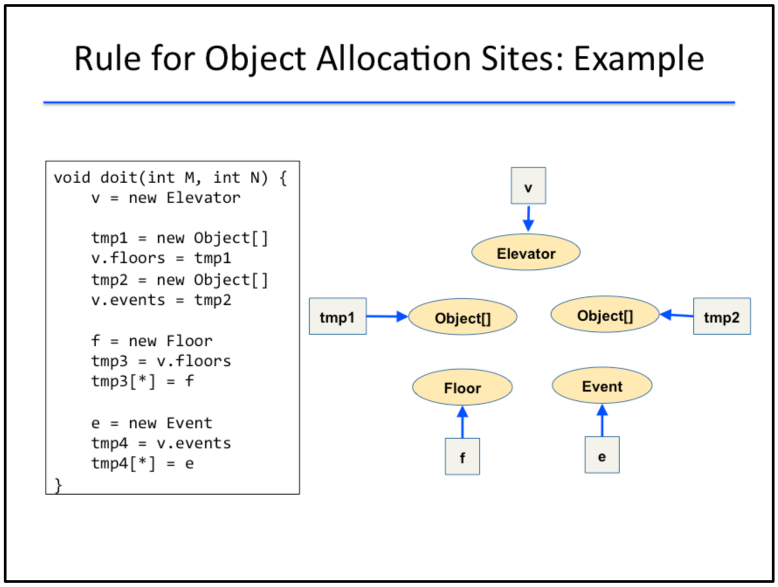
After:

By now, you should be convinced that we can take a program written in our simplified Java-like language and express its pointer operations using the six simple statements we introduced earlier.

In order to perform the chaotic iteration algorithm, we need to create rules to manipulate our points-to graph for each of these six simple statements.

Let’s look at the rule we’ll apply for the first type of statement, an object allocation. For the statement v = new B, we create a new allocation site node called B, we create a variable node for v (if it doesn’t already exist), and then add a blue arrow from the variable node to the allocation site node.

Note that if there is already an arrow from v to another allocation site node (say A), we just need to add a new arrow from v to B. This rule, as well as all the remaining rules we’ll discuss, is a “weak update,” in which we accumulate instead of replace the points-to information (which would be a “strong update”). This type of update rule is a hallmark of flow-insensitivity.



Rule for Object Allocation Sites: Example

```
void doit(int M, int N) {
    v = new Elevator

    tmp1 = new Object[]
    v.floors = tmp1
    tmp2 = new Object[]
    v.events = tmp2

    f = new Floor
    tmp3 = v.floors
    tmp3[*] = f

    e = new Event
    tmp4 = v.events
    tmp4[*] = e
}
```

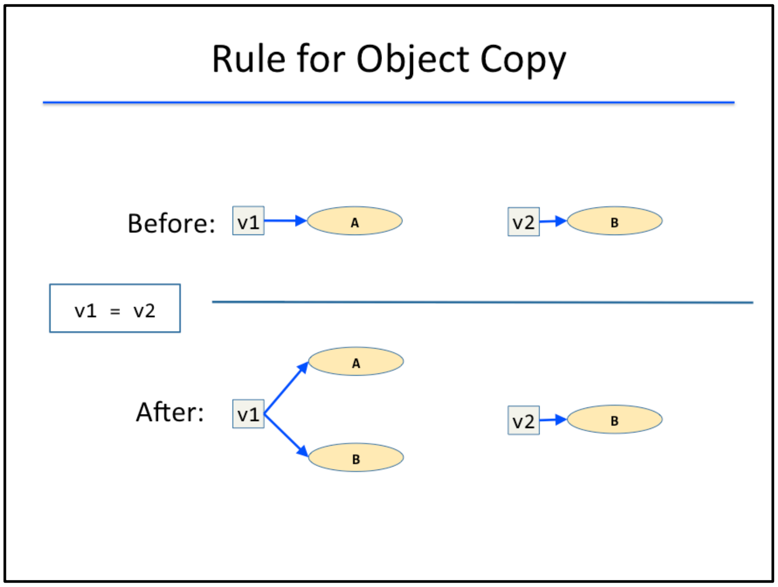
Looking at our example Elevator program, let’s apply the object allocation sites rule for each object allocation statement.

From this first allocation statement, we create an allocation site node for the Elevator, we create a variable node for v, and we make the variable node point to the allocation site node.

Similarly, we have four more allocation statements in this program:

- tmp1 points to a new Object array (point to “tmp1 = new Object[]” as appropriate nodes and arrow appear on right-hand side),
- tmp2 also points to a new Object array (point to “tmp2 = new Object[]” as appropriate nodes and arrow appear on right-hand side),
- f points to a new Floor object (point to “f = new Floor” as appropriate nodes and arrow appear on right-hand side), and
- e points to a new Event object (point to “e = new Event” as appropriate nodes and arrow appear on right-hand side).

Notice that we create separate nodes for the two Object arrays as they are allocated at different sites.



Rule for Object Copy

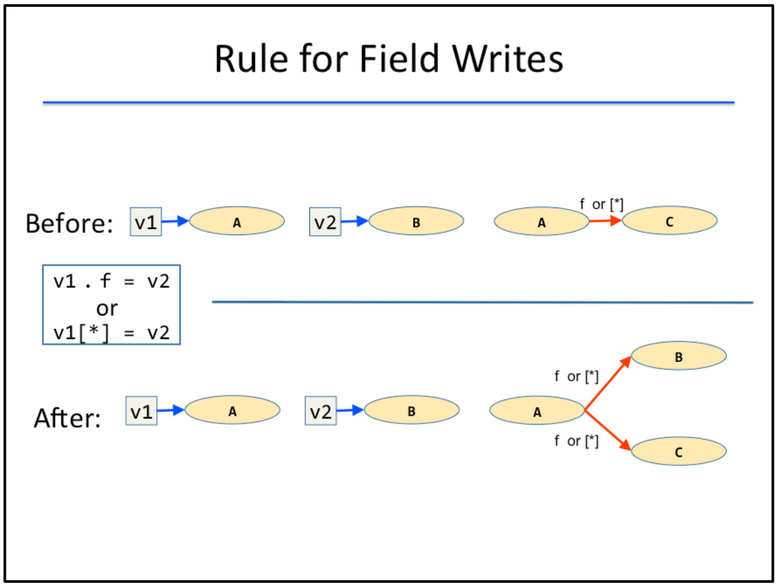
Before:

v1 = v2

After:

The next rule we'll look at is for the second type of statement in our grammar, an object copy. For the statement v1 = v2, we create a variable node for v1 (if it doesn't already exist), and then add a blue arrow from the variable node for v1 to all nodes pointed to by the variable node for v2.

Again note that we do not remove or replace any existing arrows from v1, such as this one (gesture). v1 merely accumulates another arrow to B.



Rule for Field Writes

Before:

```
v1 . f = v2
   or
v1[*] = v2
```

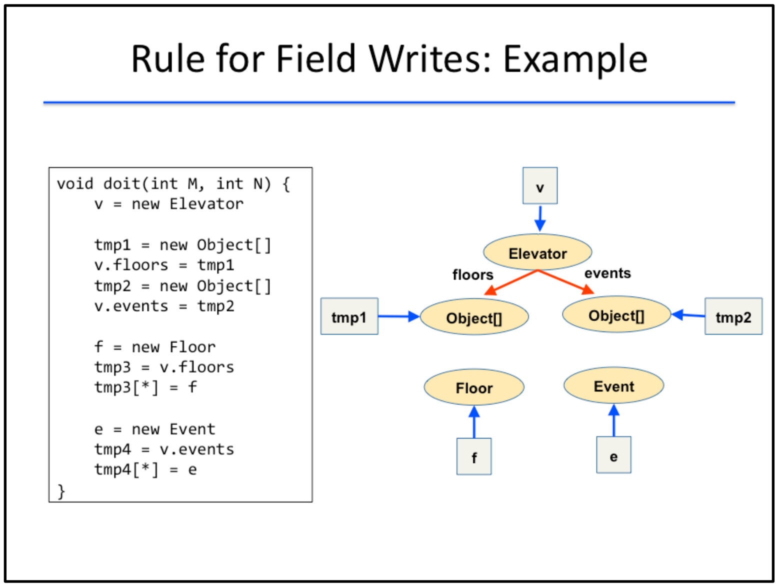
After:

Our next rule is for field-write statements of one of the following forms (gesture to the two forms).

In this case, if v1 points to A and v2 points to B, then we add a red arrow from the node for A to the node for B. We then label that arrow by the name of the field (in this case, f) or by an asterisk, if the field-write happens via an array. This reflects the fact that the field f of some object allocated at site A may point to some object allocated at site B as a result of executing this statement in some run of the containing program.

If there isn't already a node for v1 or v2, the operation of this rule amounts to skipping this statement temporarily and handling it in the next iteration.

Also, if v1 and v2 point to the same node, then the arrow we add will be an arrow from the node to itself.



Rule for Field Writes: Example

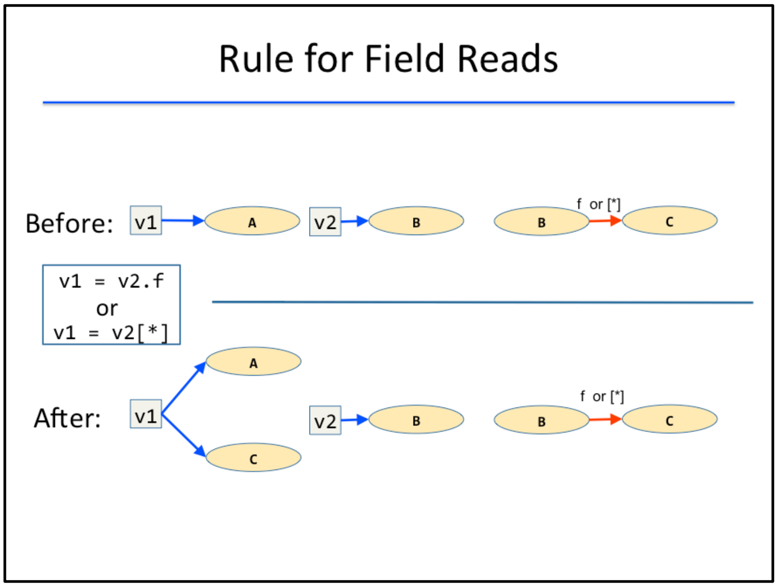
```
void doit(int M, int N) {
    v = new Elevator

    tmp1 = new Object[]
    v.floors = tmp1
    tmp2 = new Object[]
    v.events = tmp2

    f = new Floor
    tmp3 = v.floors
    tmp3[*] = f

    e = new Event
    tmp4 = v.events
    tmp4[*] = e
}
```

Now let’s apply the field-write rule to our example program.

In this first field-write highlighted here, variable tmp1 is pointing to an Object array, and variable v is pointing to an Elevator object. By the rule we just discussed, we add a red arrow from the Elevator object to the Object array, and we label it “floors” to match the name of the field being written to.

Similarly, for the second field-write highlighted here, we add a red arrow from the Elevator object to the Object array pointed to by tmp2, and we label this arrow “events” to match the name of the field.

The latter two field-writes in this program are through arrays. However, since the variables nodes for tmp3 and tmp4 haven’t been created yet, we skip these field-write operations and will come back to them in a later iteration.



Rule for Field Reads

Before:

```
v1 = v2.f
   or
v1 = v2[*]
```

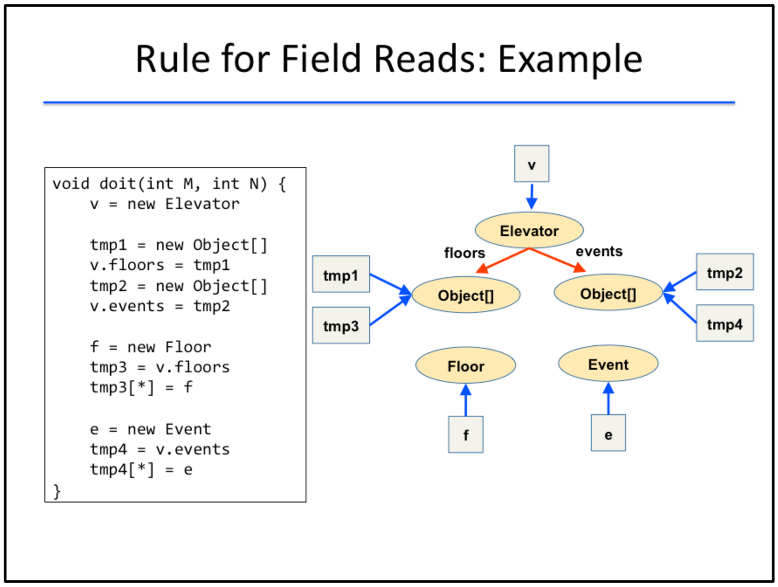
After:

Our next rule is for field-read statements of one of the following forms (gesture to the two forms).

This rule states that, if v2 points to B and B points to C via the field f or an asterisk (as appropriate), then we add a blue arrow from the node for v1 to the node for C. (If a node for v1 didn't already exist, we would create such a node before adding the blue arrow.)

Note that object B may in fact be pointing to many other objects via arrows labeled by the field in question.   In this case we would need to add an arrow from v1 to each of these nodes in order to reflect the fact that the field f, and therefore v1, may point to any one of these objects.

If there isn't already a variable node for v2 or an arrow from B to another node via f or [*], we skip the statement temporarily and try to handle it in the next iteration.



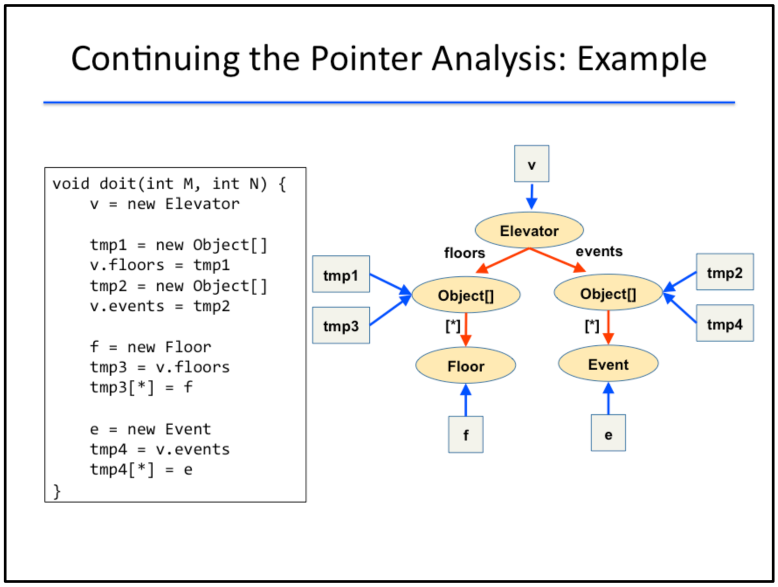
Rule for Field Reads: Example

```
void doit(int M, int N) {
    v = new Elevator

    tmp1 = new Object[]
    v.floors = tmp1
    tmp2 = new Object[]
    v.events = tmp2

    f = new Floor
    tmp3 = v.floors
    tmp3[*] = f

    e = new Event
    tmp4 = v.events
    tmp4[*] = e
}
```

Now let’s apply the field-read rule to our example program.

In this first field-read, we create a node for the variable tmp3. Then we add a blue arrow from tmp3 to the Object array which the “floors” field of the Elevator points to.

Similarly, we create a node for the variable tmp4, and then we add add a blue arrow from tmp4 to the Object array which the “events” field of the Elevator points to.

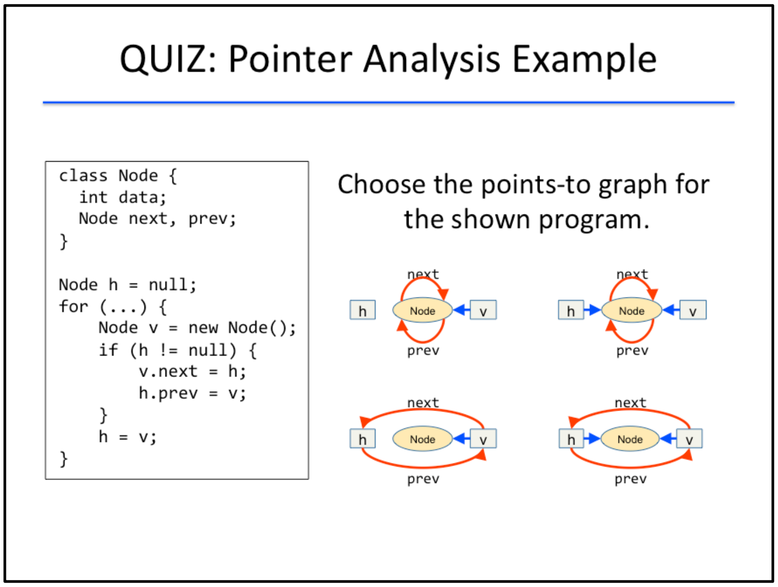


Continuing the Pointer Analysis: Example

```
void doit(int M, int N) {
    v = new Elevator

    tmp1 = new Object[]
    v.floors = tmp1
    tmp2 = new Object[]
    v.events = tmp2

    f = new Floor
    tmp3 = v.floors
    tmp3[*] = f

    e = new Event
    tmp4 = v.events
    tmp4[*] = e
}
```

Now that we've created nodes for tmp3 and tmp4, we can apply the field-write rules to the statements we skipped previously. Looking at this field-write statement (tmp3[*] = f), since tmp3 points to this Object array and f points to this Floor node, we can add a red arrow labeled by an asterisk from the Object array to the Floor node. Then, for this other field-write statement (tmp4[*] = e), we likewise add a red arrow labeled by an asterisk from the Object array pointed at by tmp4 to the Event node pointed at by e.

At this point, iterating through the set of statements again will produce no changes to the graph, so the chaotic iteration algorithm will terminate, leaving us with the points-to graph we see here.



QUIZ: Pointer Analysis Example

```
class Node {
  int data;
  Node next, prev;
}

Node h = null;
for (...) {
    Node v = new Node();
    if (h != null) {
        v.next = h;
        h.prev = v;
    }
    h = v;
}
```

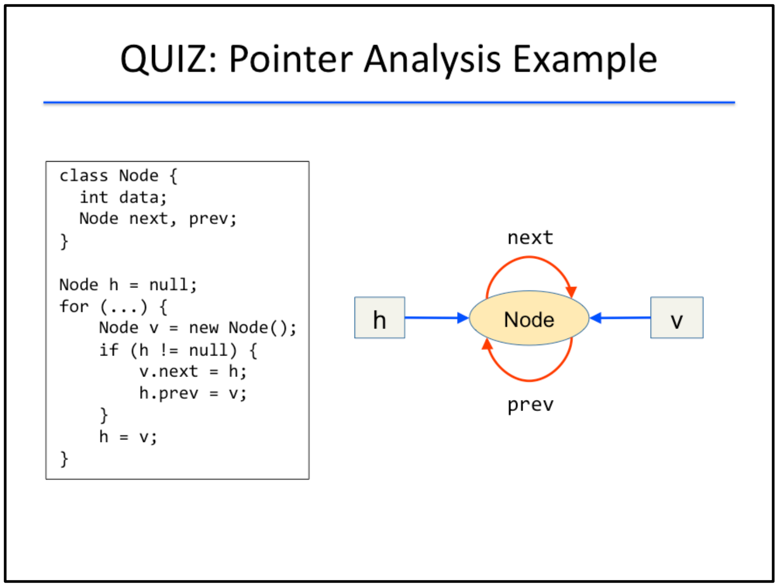
Choose the points-to graph for the shown program.

{QUIZ SLIDE}

Now that we’ve looked at all the rules for pointer analysis, let’s return to an example we saw at the beginning of the lesson. Here we have some code for creating a doubly linked list of Node objects.

For this quiz, transform the code on the left-hand side using the flow-insensitivity approximation scheme, and then perform the chaotic iteration algorithm yourself on the resulting set of statements. What does the points-to graph look like when the chaotic iteration algorithm concludes?

Click the radio button corresponding to the graph that represents the correct points-to graph.



QUIZ: Pointer Analysis Example

```
class Node {
  int data;
  Node next, prev;
}

Node h = null;
for (...) {
    Node v = new Node();
    if (h != null) {
        v.next = h;
        h.prev = v;
    }
    h = v;
}
```

{SOLUTION SLIDE}

The answer to this quiz is the top-right graph.

Let's work through the example together and see how the chaotic iteration algorithm gets us here.

First, we transform the code by applying the flow-insensitivity approximation, which results in the following four highlighted statements (highlight relevant statements from code). Note that we didn't highlight the statement "h = null". Remember that pointer analysis is a weak update analysis, so we never remove an edge from the graph once we add it in. Instead, the statement "h = null" just tells us to add no new edges to the node for variable h. So we can safely ignore this statement altogether in our analysis.

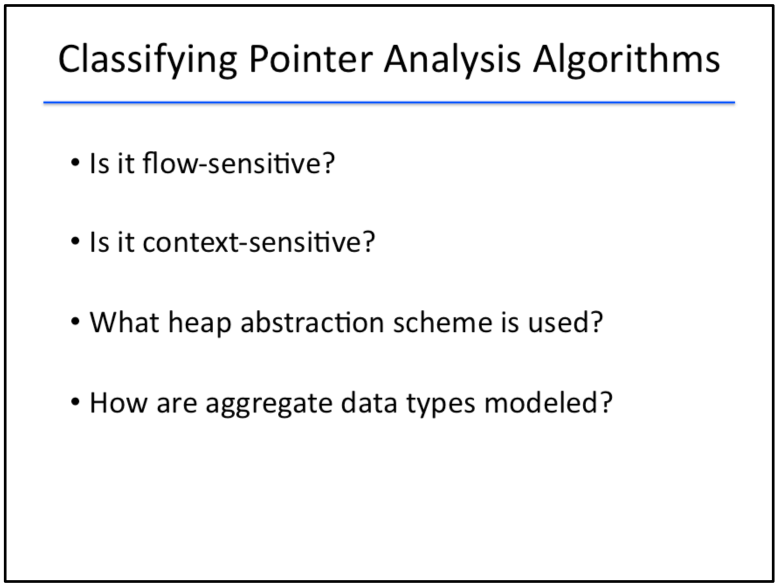
Now let's proceed through the highlighted statements, one statement at a time, and build the points-to graph iteratively.

The first statement we will look at is "v = new Node". Applying the rule for object allocation, we create a variable node for v, an allocation site node called "Node", and a blue arrow from v to "Node" (Node, v, and blue arrow from v to Node appear).

Let's now look at the statement "h = v" (remember that we can iterate through the statements in any order, since the statements are coming from an unordered set). Applying the rule for this statement, we create a variable node for h and a blue arrow from h to the Node object that v points to (h and blue arrow from h to Node appear).

Now let's apply the rule for the field-write statement "v.next = h". Since both h and v point to the Node object, we create a red edge from the Node object to itself and label it "next" (add red arrow from Node to itself labeled "next"). Similarly, for the field-write statement "h.prev = v", we create a red edge from the Node object to itself and label it "prev" (add red arrow from Node to itself labeled "prev").

Iterating through the set of statements again doesn't cause us to change the graph at all, so the chaotic iteration algorithm terminates, leaving us with the graph we see here as our points-to graph.



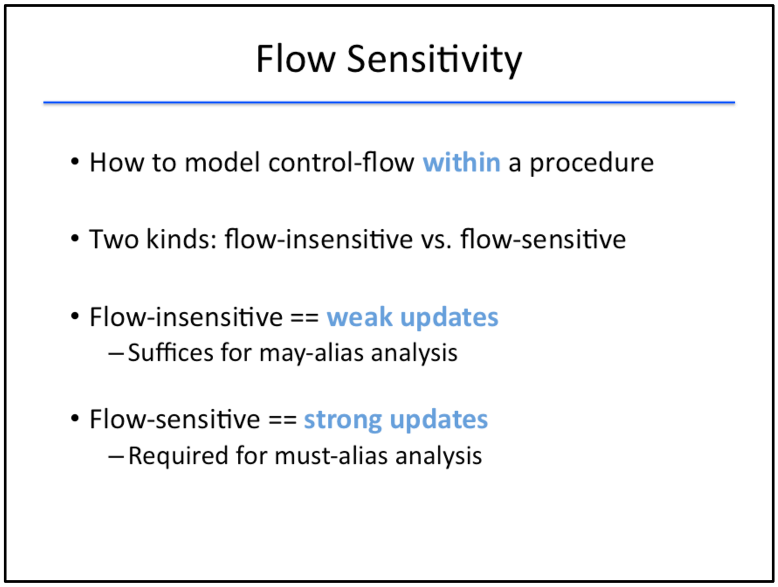
Classifying Pointer Analysis Algorithms

- Is it flow-sensitive?
- Is it context-sensitive?
- What heap abstraction scheme is used?
- How are aggregate data types modeled?

Thus far, we have looked at a particular pointer analysis algorithm that is commonly used. Recall from the beginning of this lesson that there are many different algorithms for pointer analysis with varying precision. These algorithms can be classified along many dimensions.

Let's look at four of the most important dimensions, and see how the pointer analysis algorithm we learnt fits in this classification.

The four dimensions are: whether or not it is a flow sensitive analysis, whether or not it is a context sensitive analysis, the kind of heap abstraction scheme that it uses, and how it models aggregate data types such as arrays and structures.



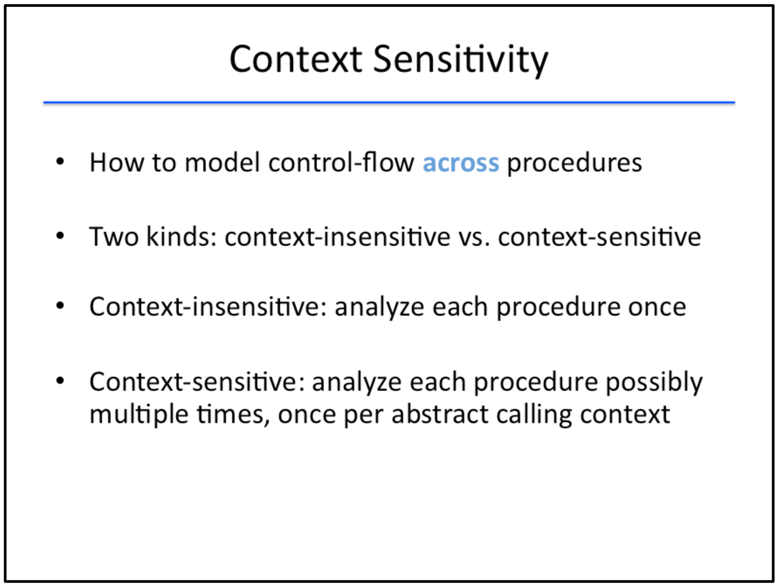
Flow Sensitivity

- How to model control-flow **within** a procedure
- Two kinds: flow-insensitive vs. flow-sensitive
- Flow-insensitive == **weak updates**
  - Suffices for may-alias analysis
- Flow-sensitive == **strong updates**
  - Required for must-alias analysis

Flow-sensitivity concerns how a pointer analysis algorithm models control-flow within a procedure or function, called intra-procedural control-flow. Pointer analysis algorithms can be broadly classified into two kinds, flow-insensitive and flow-sensitive, based on how they handle intra-procedural control-flow.

Flow-insensitive pointer analysis algorithms, like the one we just learnt, ignore control-flow entirely, viewing the program as an unordered set of statements. A hallmark of flow-insensitive analyses is that these analyses only generate new facts as they progress; they never kill any previously generated facts. We observed this in the case of the pointer analysis algorithm we just saw, wherein the points-to graph only grew in size as each statement of the program was considered. We say that such algorithms perform *weak updates*. Such algorithms typically suffice for may-alias analysis, that is, it is practical for a may-alias analysis to have a low false positive rate despite being flow-insensitive.

Flow-sensitive pointer analysis algorithms, on the other hand, are capable of killing facts in addition to generating facts. We say that such algorithms perform *strong updates*. Such algorithms are typically required for must-alias analysis, that is, it is impractical for a must-alias analysis to have a low false positive rate by being flow-insensitive.



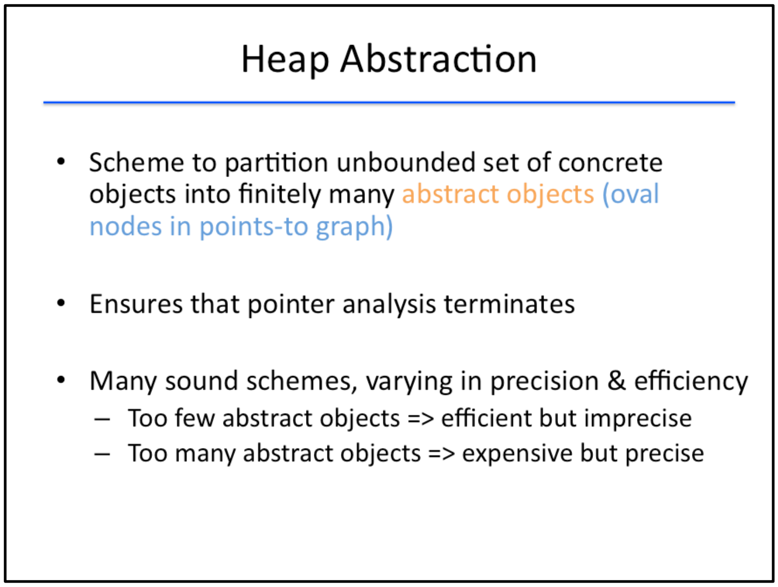
Context Sensitivity

- How to model control-flow **across** procedures
- Two kinds: context-insensitive vs. context-sensitive
- Context-insensitive: analyze each procedure once
- Context-sensitive: analyze each procedure possibly multiple times, once per abstract calling context

Another common dimension for classifying pointer analysis algorithms is context sensitivity, which concerns how to handle control-flow across procedures, called inter-procedural control-flow. Pointer analysis algorithms can be broadly classified into two kinds, context-insensitive and context-sensitive, based on how they handle inter-procedural control-flow.

Context-insensitive pointer analysis algorithms analyze each procedure once, regardless of how many different parts of the program call that procedure. These algorithms are relatively imprecise, as they conflate together aliasing facts that arise from different calling contexts. But they are very efficient, since they analyze each procedure only once.

Context-sensitive pointer analysis algorithms, on the other hand, potentially analyze each procedure multiple times, once per abstract calling context. These algorithms are relatively precise but expensive. They differ primarily in the manner in which they abstract the calling context. There are many different schemes for abstracting the calling context that we will study in the next lesson. The choice of the scheme is dictated by the desired tradeoff between precision and efficiency for a client of the pointer analysis.



Heap Abstraction

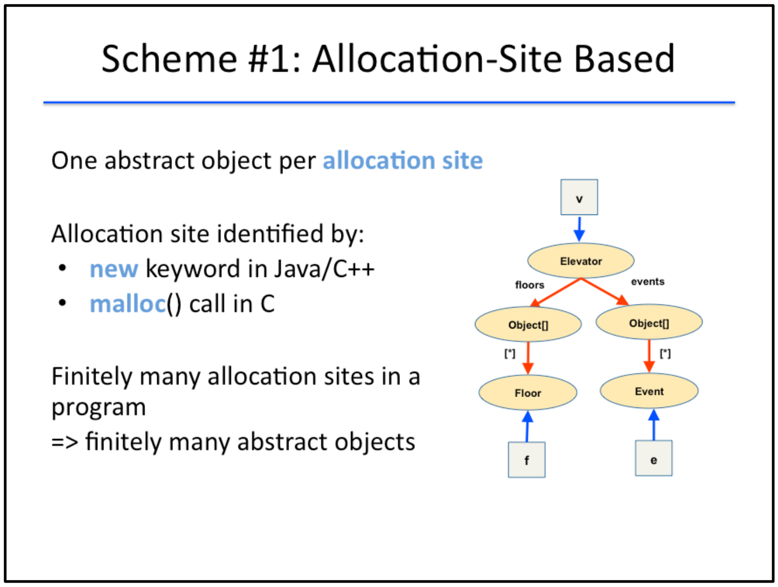
- Scheme to partition unbounded set of concrete objects into finitely many abstract objects (oval nodes in points-to graph)
- Ensures that pointer analysis terminates
- Many sound schemes, varying in precision & efficiency
  - Too few abstract objects => efficient but imprecise
  - Too many abstract objects => expensive but precise

While flow sensitivity and context sensitivity concern abstracting control flow, the heap abstraction concerns abstracting program data, in particular, dynamically allocated objects, which we call the heap.

The heap abstraction scheme specifies how to partition an unbounded set of concrete objects that the program may create into finitely many abstract objects. Each abstract object corresponds to an oval node in the points-to graph. This partitioning is at the heart of ensuring that pointer analysis terminates.

Much like any of the abstractions we have seen in this course, designing a suitable heap abstraction is an art. Many sound heap abstraction schemes exist with varying precision and efficiency. As a rule of thumb, a scheme that produces too few abstract objects in the points-to graph will result in an efficient but imprecise pointer analysis -- imprecise because it conflates more concrete objects into each of the few abstract objects. On the other hand, a scheme that produces too many abstract objects in the points-to graph will result in an expensive but precise pointer analysis.

Let’s take a look at three heap abstraction schemes that are commonly used.



Scheme #1: Allocation-Site Based

One abstract object per **allocation site**

Allocation site identified by:

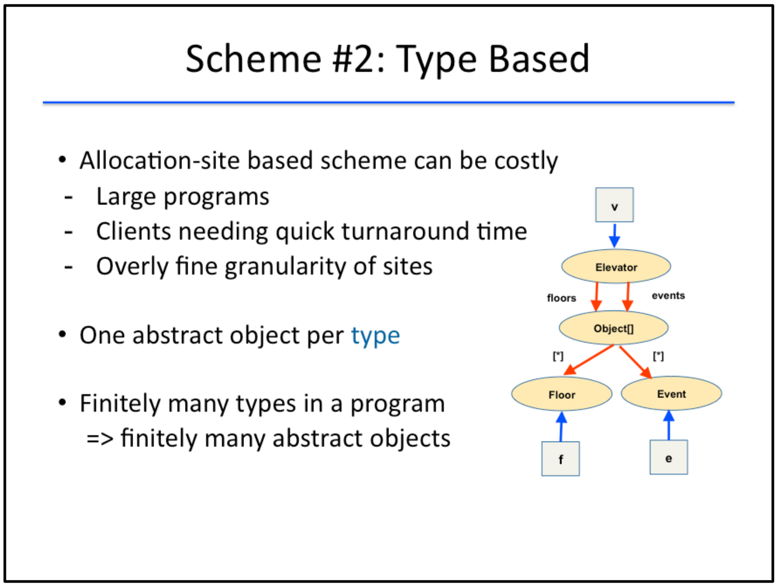
- **new** keyword in Java/C++
- **malloc**() call in C

Finitely many allocation sites in a program
=> finitely many abstract objects

The heap abstraction scheme used by the pointer analysis algorithm we studied in this lesson is called the allocation-site based scheme. This scheme partitions concrete objects based on the site in the program where they are created, called the allocation site. In other words, each abstract object under this scheme corresponds to a separate allocation site.

Allocation sites are identified by the new keyword in Java and C++, and by the malloc function call in C. Since there are finitely many allocation sites in a program, a pointer analysis using this scheme is guaranteed to result in a finite number of abstract objects in the constructed points-to graph.

Here is the points-to graph that was constructed using the allocation-site based scheme for our example elevator program. Notice that each of the five ovals corresponds to a separate allocation site in the program.



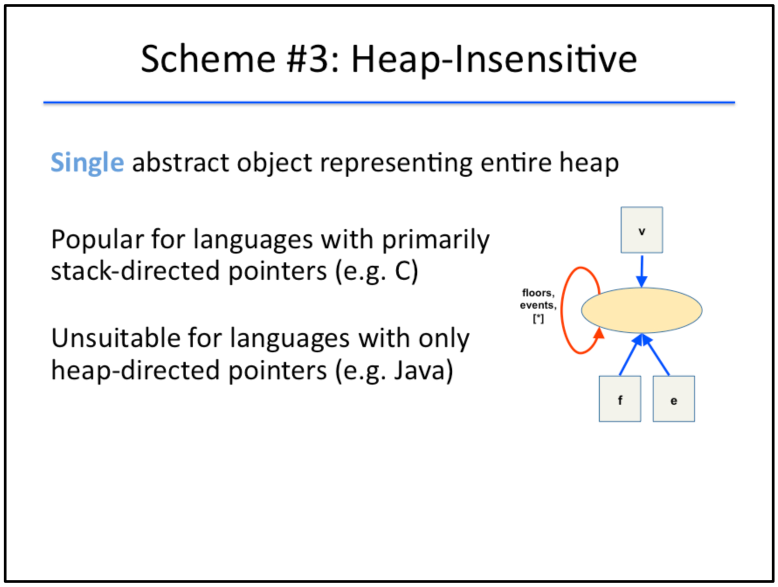
Scheme #2: Type Based

- Allocation-site based scheme can be costly
  - Large programs
  - Clients needing quick turnaround time
  - Overly fine granularity of sites
- One abstract object per type
- Finitely many types in a program
  => finitely many abstract objects

Although the number of allocation sites in any program is finite, tracking a separate abstract object per allocation site can be prohibitively expensive for large programs, which can contain too many allocation sites, or for clients of pointer analysis that need a quick turnaround time to aliasing queries, such as an integrated development environment, or for situations when it is unnecessary to make distinctions at the fine granularity of allocation sites.

The type-based scheme is a cheaper scheme that can be used in such situations. This scheme partitions concrete objects based on their type instead of based on the site where they are created. In other words, each abstract object under this scheme corresponds to a separate type. Since there are finitely many types in a program, a pointer analysis using this scheme is guaranteed to result in a finite number of abstract objects in the constructed points-to graph.

Here is the points-to graph that would be computed for our example elevator program by a pointer analysis using the type-based scheme. Notice that each of the four ovals corresponds to a separate type in the program, which is indicated in the oval. In particular, notice that the two separate ovals that were created for the arrays of floors and events are now conflated into one, as both have the same type: an array of objects.



Scheme #3: Heap-Insensitive

Single abstract object representing entire heap

Popular for languages with primarily stack-directed pointers (e.g. C)

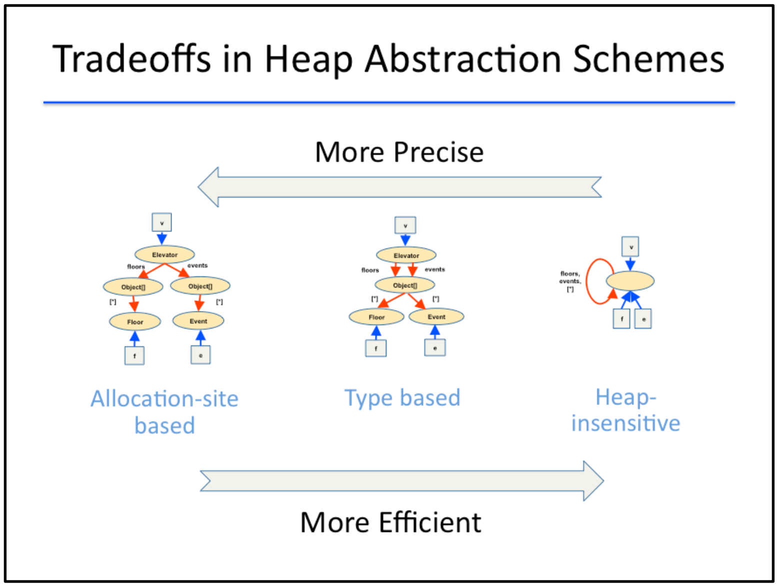
Unsuitable for languages with only heap-directed pointers (e.g. Java)

Yet another heap abstraction scheme is one that does not make any distinctions between dynamically allocated objects, which we will call the heap-insensitive scheme. This scheme uses a single abstract object to model the entire heap.

Here is the points-to graph that would be computed for our example elevator program by a pointer analysis using this scheme. Notice that it contains a single oval representing the entire heap. Looking at this points-to graph, it should be easy to see that this scheme is highly imprecise for reasoning about the heap but it is sound nevertheless.

So you might wonder: are there scenarios in which this scheme is useful? The answer is yes, for languages with primarily stack-directed pointers like C, where malloc calls are sparse. Pointer analyses for such languages derive most of their precision by reasoning about stack-directed pointers, and completely ignoring heap-directed pointers.

Of course, this scheme is unsuitable for languages with only heap-directed pointers like Java.



Tradeoffs in Heap Abstraction Schemes

More Precise

Allocation-site based

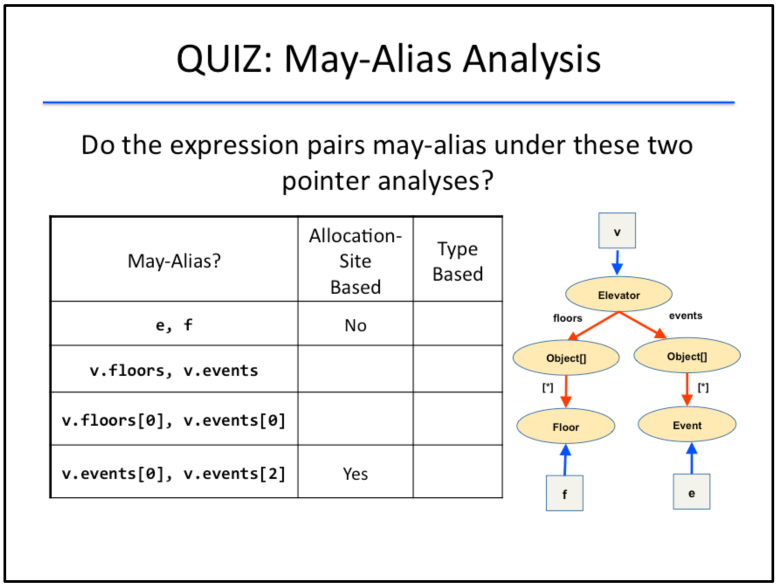
Type based

Heap-insensitive

More Efficient

To recap, the three different heap abstraction schemes that we just saw strike a different tradeoff between precision and efficiency. As we go from the allocation-site based scheme through the type-based scheme to the heap-insensitive scheme, the pointer analysis gets more efficient, but also less precise.

It is important to remember that these are not the only three schemes: there are many other schemes in the literature, and you can even design your own depending on your analysis needs. For instance, there are many schemes that are more expensive and precise than the allocation-site based scheme; these schemes can even make distinctions between objects created at the same allocation site. At the same time, remember that we can never have a scheme that will allow pointer analysis to make distinctions between all concrete objects and terminate in finite time.



QUIZ: May-Alias Analysis

Do the expression pairs may-alias under these two pointer analyses?

May-Alias?	Allocation-Site Based	Type Based
`e, f`	No	
`v.floors, v.events`		
`v.floors[0], v.events[0]`		
`v.events[0], v.events[2]`	Yes	

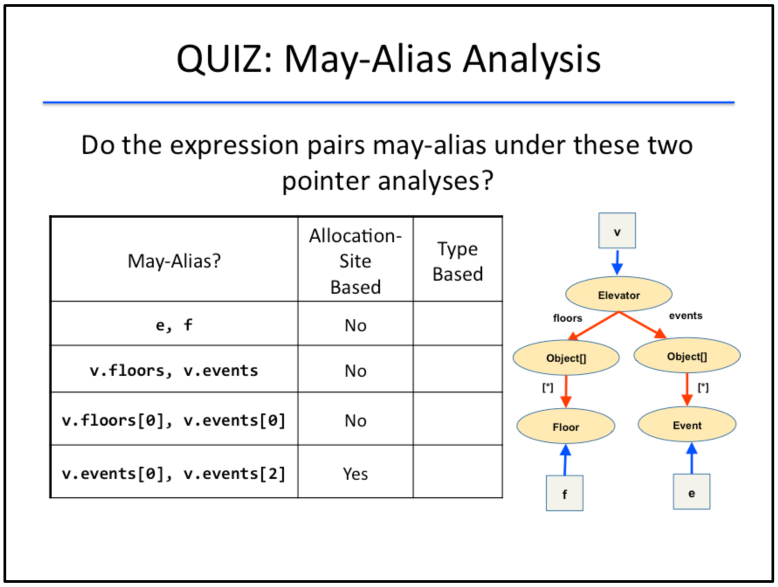
v
Elevator
floors
events
Object[]
Object[]
[*]
[*]
Floor
Event
f
e

{QUIZ SLIDE}

Here's the points-to graph we constructed using allocation site-based pointer analysis. If we ask the question, "MAY the pointers e and f alias?", then we would answer NO (No appears), because the arrows from e to its referred abstract object and f to its referred abstract object are different nodes in the points-to graph. Being in different nodes reflects the fact that the concrete objects referred to by e and f are allocated in distinct places in memory. Thus e and f cannot be aliases.

On the other hand, if we ask whether v.events[0] and v.events[2] may-alias, we answer YES (Yes appears). If we follow the arrows in the graph to find the abstract object referred to by v.events[0], we would first follow the blue arrow to the Elevator node, then the red "events" arrow to this (gesture) Object array node, and then the red asterisk arrow to the Event node. If we follow the arrows in the graph for v.events[2], we end up following the same path to end in the same node. Thus, according to this graph, it could be the case that v.events[0] and v.events[2] point to the same concrete object, so we cannot prove they are not aliases. Note that the concrete objects could be different, but we cannot say one way or the other using this heap abstraction scheme.

Now, fill in the remaining boxes in this table, answering the question "MAY the two given pointers alias?" under both the allocation site-based and type-based heap abstractions. You can see the type-based points-to graph we presented earlier in the instructor notes.



QUIZ: May-Alias Analysis

Do the expression pairs may-alias under these two pointer analyses?

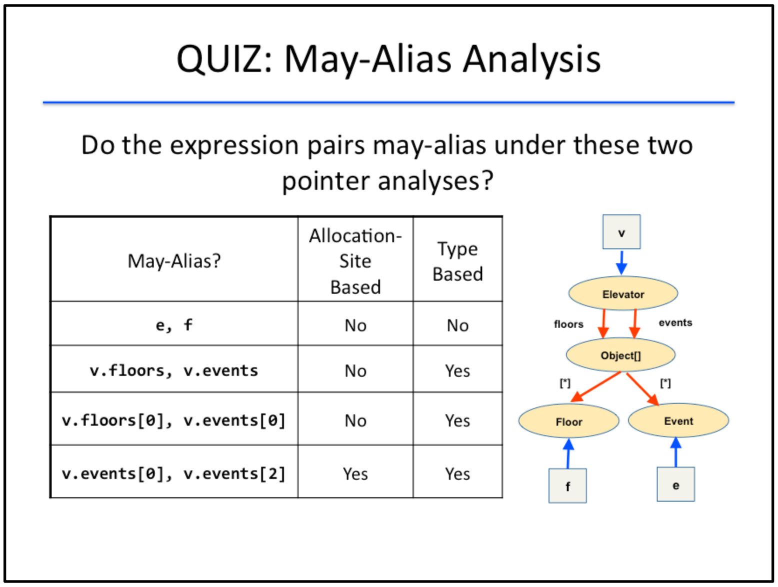
May-Alias?	Allocation-Site Based	Type Based
`e, f`	No	
`v.floors, v.events`	No	
`v.floors[0], v.events[0]`	No	
`v.events[0], v.events[2]`	Yes	

{SOLUTION SLIDE 1}

Let's start by filling in the remaining cells of the allocation site-based column.

For v.floors, we follow the arrow from v to Elevator and then the arrow labeled "floors" to this (gesture) Object array node. For v.events, we follow the arrow from v to Elevator and then the arrow labeled "events" to this (gesture) Object array node. Since these two nodes are distinct in the points-to graph, v.floors and v.events always refer to concrete objects allocated at different sites in the program. Thus they cannot refer to the same concrete object, so the question "MAY v.floors and v.events alias?" is answered "NO" ("No" appears in first blank cell of Allocation column).

For v.floors[0] and v.events[0], we need to follow one more arrow each than we did when analyzing v.floors and v.events. We follow the arrows labeled by the asterisks. Since we end up at the Floor node for v.floors[0] and the Event node for v.events[0], we can again conclude that v.floors[0] and v.events[0] always refer to different concrete objects. So the question "MAY they alias?" is answered "NO" ("No" appears in second blank cell of Allocation column).



QUIZ: May-Alias Analysis

Do the expression pairs may-alias under these two pointer analyses?

May-Alias?	Allocation-Site Based	Type Based
`e, f`	No	No
`v.floors, v.events`	No	Yes
`v.floors[0], v.events[0]`	No	Yes
`v.events[0], v.events[2]`	Yes	Yes

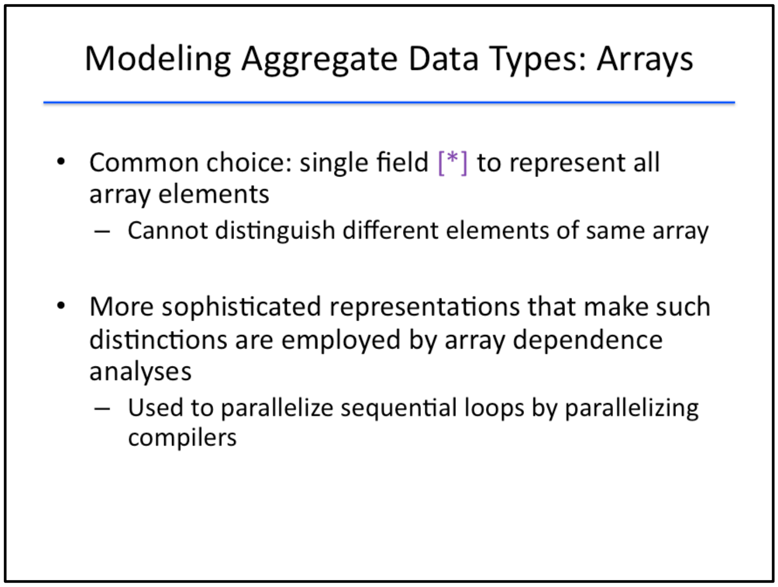
{SOLUTION SLIDE 2}

Now let's look at the points-to graph the chaotic iteration algorithm generates for the type-based heap abstraction. As before in the allocation site-based abstraction, e and f point to two different nodes. This means that we can be certain e and f point to objects of a different type, so we can answer the question "MAY e and f alias?" by "NO" ("No" appears in first blank cell of Type-Based column).

However, we see a difference in the precision of these two analyses when asking whether v.floors and v.events may alias. Following the arrows from v to the Elevator node and then the "floors" arrow from the Elevator node, we reach the Object array node where v.floors points. But if we'd taken the "events" arrow from Elevator, we'd end up at the same Object array node. So the type-based heap abstraction scheme cannot distinguish that these two pointers don't actually alias in our example program. It answers "YES" to the question of whether they MAY alias ("Yes" appears in second blank cell of Type-Based column).

Now, suppose we wanted to know whether v.floors[0] and v.events[0] MAY alias. We first follow the arrows to the Object array node that v.floors and v.events lead us to. Now we need to follow the arrow labeled by an asterisk, but there are two such arrows leading away from the node. We need to follow both in parallel: all this graph tells us is that v.floors[0] could point to either the Floor node or the Event node. And it tells us the same thing for v.events[0]: this pointer could refer to either the Floor node or the Event node. Since the set of nodes that v.floors[0] could point to and the set of nodes that v.events[0] could point to overlap, our analysis cannot prove that they do not point to the same object. So we answer the "MAY-alias" question "YES" ("Yes" appears in third blank cell of Type-Based column).

Finally, let's look at whether v.events[0] and v.events[2] MAY alias. Again, following the paths through the points-to graph, v.events points to the Object array node. And again, since we now have an array subscript to dereference, we take all arrows from this node labeled by an asterisk in parallel. So, according to this graph, v.events[0] could point to either the Floor node or the Event node. And v.events[2] could also point to either the Floor or Event node. Since the sets of nodes they could point to overlap, we cannot prove using this heap abstraction that v.events[0] and v.events[2] do not alias. So we answer "YES" to the question of whether they MAY alias ("Yes" appears in last blank cell of Type-Based column).



Modeling Aggregate Data Types: Arrays

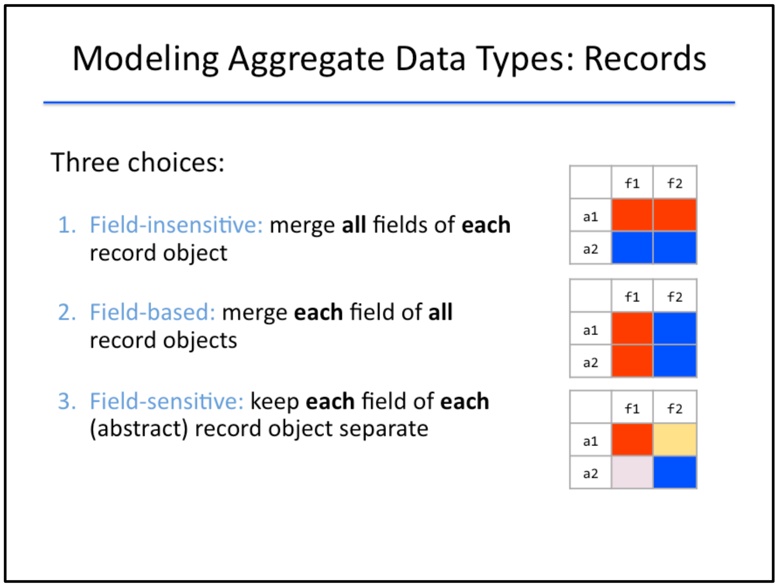
- Common choice: single field [*] to represent all array elements
  - Cannot distinguish different elements of same array
- More sophisticated representations that make such distinctions are employed by array dependence analyses
  - Used to parallelize sequential loops by parallelizing compilers

Another dimension for abstracting program data in a pointer analysis concerns how to model aggregate data types. These include arrays and records. Let's look at arrays first.

A common choice in pointer analyses is to use a single field, denoted by asterisk, to represent all elements of an array. Such analyses therefore cannot distinguish between different elements of an array. The pointer analysis algorithm we saw earlier in this lesson uses this representation.

More sophisticated representations make distinctions between different elements of an array. Such representations are employed in so-called array dependence analyses whose primary goal is to determine whether two integer expressions refer to the same element of an array, much like the primary goal of pointer analysis is to determine whether two pointer expressions alias.

Array dependence analysis in turn is used in parallelizing compilers to parallelize sequential loops.



Modeling Aggregate Data Types: Records

Three choices:

1. Field-insensitive: merge **all** fields of **each** record object

	f1	f2
a1	🟥	🟥
a2	🟦	🟦

2. Field-based: merge **each** field of **all** record objects

	f1	f2
a1	🟥	🟦
a2	🟥	🟦

3. Field-sensitive: keep **each** field of **each** (abstract) record object separate

	f1	f2
a1	🟥	🟨
a2	🟪	🟦

Now let's look at the another common kind of aggregate data type: records.

Records are also known as structs in C or classes in object-oriented languages like C++ and Java.

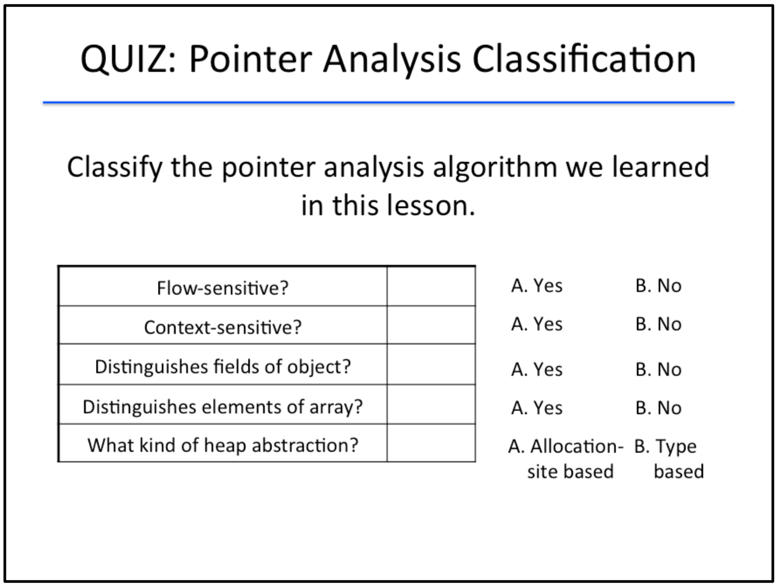
There are three common choices that pointer analyses use to model records.

We will illustrate these three choices using a record type with two fields f1 and f2. Suppose there are two objects a1 and a2 of this record type.

Then, one choice is to use a field-insensitive representation, which merges all fields of each record object. In other words, this representation prevents the analysis from distinguishing between different fields, such as f1 and f2, of the same record object, such as a1 or a2.

Another choice is to use a so-called field-based representation, which merges each field across all record objects. In other words, this representation prevents the analysis from distinguishing between the same field, such as f1 or f2, of different objects, such as a1 and a2.

The third choice is to use a field-sensitive representation, which keeps each field of each abstract record object separate. Assuming that a1 and a2 denote distinct abstract objects, this representation allows the analysis to distinguish all four memory locations denoted by this table: field f1 of a1, field f2 of a1, field f1 of a2, and field f2 of a2. It is easy to see that this is the most precise of the three representations. The pointer analysis algorithm we studied earlier in this lesson used this representation.



QUIZ: Pointer Analysis Classification

Classify the pointer analysis algorithm we learned in this lesson.

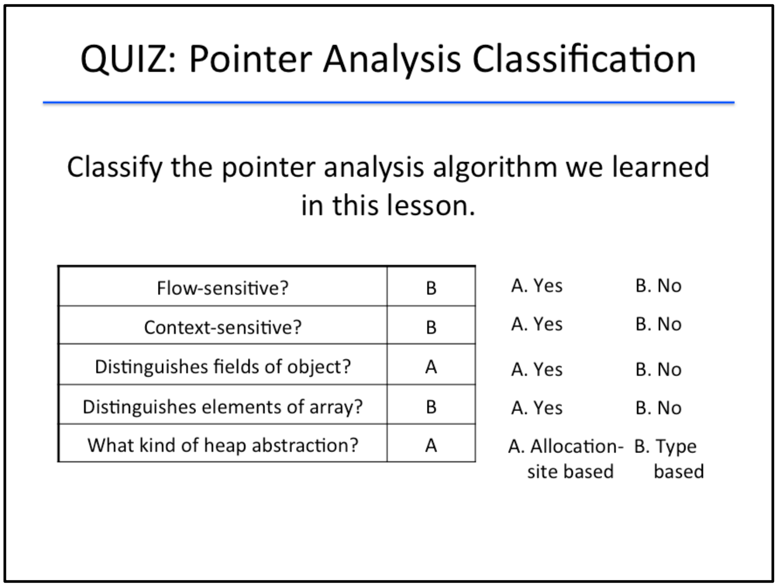
Flow-sensitive?		A. Yes	B. No
Context-sensitive?		A. Yes	B. No
Distinguishes fields of object?		A. Yes	B. No
Distinguishes elements of array?		A. Yes	B. No
What kind of heap abstraction?		A. Allocation-site based	B. Type based

{QUIZ SLIDE}

Before we close this lesson, let's reflect on the pointer analysis algorithm we looked at during the lesson. Classify it according to the dimensions we just discussed. In particular:

- Is the pointer analysis we discussed flow-sensitive?
- Is it context-sensitive?
- Did it distinguish between different fields of an obect?
- Did it distinguish between different elements of an array?
- And what kind of heap abstraction did it use: allocation site-based or type-based?

Type in A to mean "Yes" and B to mean "No." For the last question, type "A" to mean "Allocation site-based" and "B" for "Type-based".



QUIZ: Pointer Analysis Classification

Classify the pointer analysis algorithm we learned in this lesson.

Flow-sensitive?	B	A. Yes	B. No
Context-sensitive?	B	A. Yes	B. No
Distinguishes fields of object?	A	A. Yes	B. No
Distinguishes elements of array?	B	A. Yes	B. No
What kind of heap abstraction?	A	A. Allocation-site based	B. Type based

{SOLUTION SLIDE}

Now let's review the answers.

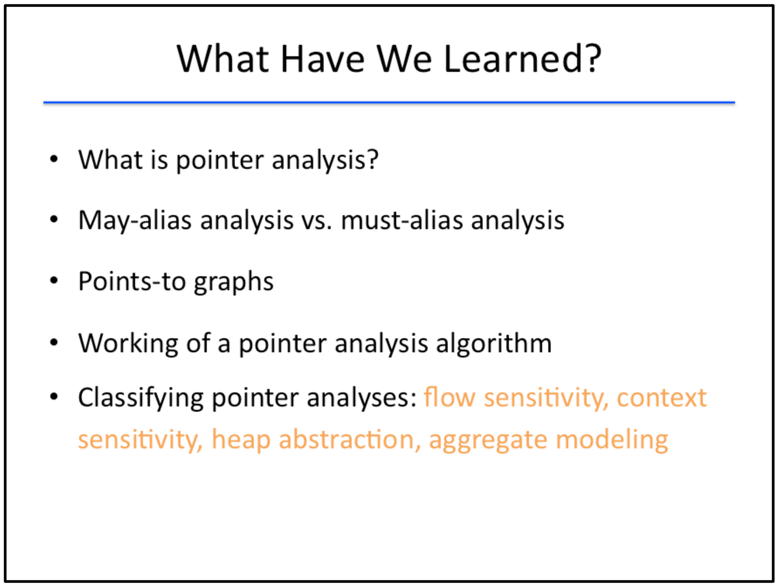
Is the pointer analysis we discussed flow-sensitive? No, we used flow insensitivity as an approximation scheme to eliminate the complexity of tracking the points-to graph at each program point.

Is the pointer analysis context-sensitive? No. While the example program we used only had a single function, the chaotic iteration algorithm would only analyze each function one time without regard to its context.

Did the pointer analysis distinguish different object fields? Yes, the algorithm did distinguish differently named object fields, such as the floors and events fields of the Elevator.

Did the pointer analysis distinguish different elements of an array? No, the algorithm abstracted away array elements using an asterisk to represent all subscripts.

And what kind of heap abstraction did the pointer analysis use: allocation site-based or type-based? Though we discussed type-based abstraction later in the lesson, the chaotic iteration algorithm we discussed used an allocation site-based heap abstraction.



What Have We Learned?

- What is pointer analysis?
- May-alias analysis vs. must-alias analysis
- Points-to graphs
- Working of a pointer analysis algorithm
- Classifying pointer analyses: flow sensitivity, context sensitivity, heap abstraction, aggregate modeling

Let's recap the main topics that we have covered in this lesson.

You have seen, at a high level, what pointer analysis is: a procedure for proving facts of the form "pointer X and pointer Y do not alias."

You saw the difference between a MAY-alias analysis and a MUST-alias analysis. Since MAY-alias analysis is less technically demanding and typically more useful in practice, it is what we refer to when we say "pointer analysis."

You saw what a points-to graph is and how it serves as an approximation of the actual state of data during a run of the program. This approximation allows us to conclude a pointer analysis in finite time at the cost of some number of false positives. You also used the points-to graph to decide whether two pointers MAY alias one another.

You saw how a particular pointer analysis algorithm---the chaotic iteration algorithm---works to build a points-to graph for a program.

And you learned different dimensions along which pointer analysis algorithms may vary, including whether they remain sensitive to control flow and calling context, the kind of heap abstraction used, and how aggregate data is modeled (for example, in arrays).

With this knowledge, you should be able to implement a pointer analysis algorithm for a simple programming language, and you should be able to determine the costs and benefits of different types of pointer analyses.

In this lesson, we presumed that the program we are analyzing consists of a single procedure. Realistic programs, on the other hand, consist of many procedures calling one another. Therefore, any realistic dataflow analysis must be able to reason about programs with multiple procedures. Many different approaches exist to extend a dataflow analysis from single-procedure programs to programs with multiple procedures. In the next lesson, we will learn about these approaches. Following the theme we have seen in this lesson and the previous one, all of these approaches will be sound but they will strike different tradeoffs between precision and efficiency; which approach to use thus depends on the desired tradeoff for the analysis you are designing.