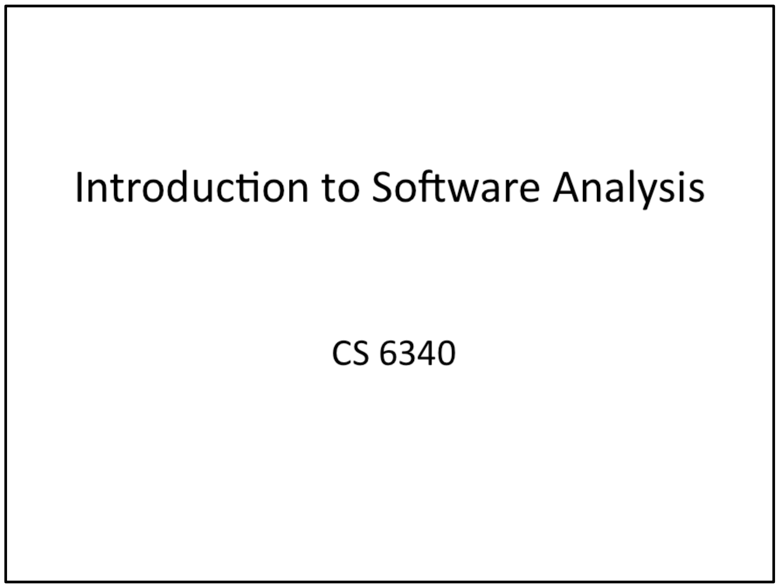


Introduction to Software Analysis

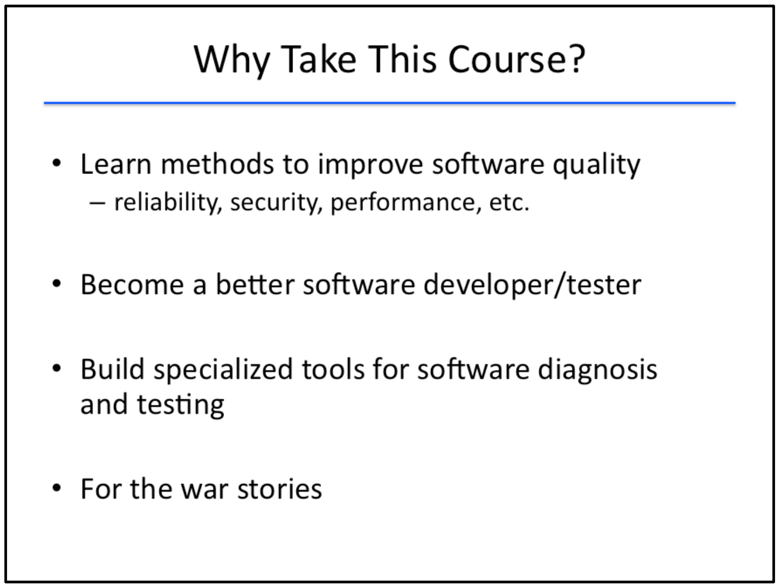
CS 6340

{HEADSHOT: Intro to entire course}

Hello students. I'm Mayur Naik, a professor of Computer Science at Georgia Tech. Welcome to Software Analysis and Testing.

In this course, we will be diving deep into the theory and practice of software analysis, which lies at the heart of many software development processes such as diagnosing bugs, testing, debugging, and more.

What this class won't do is teach you basic concepts of programming. Instead, through a mixture of basic and advanced exercises and examples, you will learn techniques and tools to enhance your existing programming skills and build better software.



Why Take This Course?

- Learn methods to improve software quality
  - reliability, security, performance, etc.
- Become a better software developer/tester
- Build specialized tools for software diagnosis and testing
- For the war stories

So why should you take this course?

** Visual for Bill Gates Quote **

Bill Gates once said and I quote “We have as many testers as we have developers. And testers spend all their time testing, and developers spend half their time testing. We're more of a testing, a quality software organization than we're a software organization."

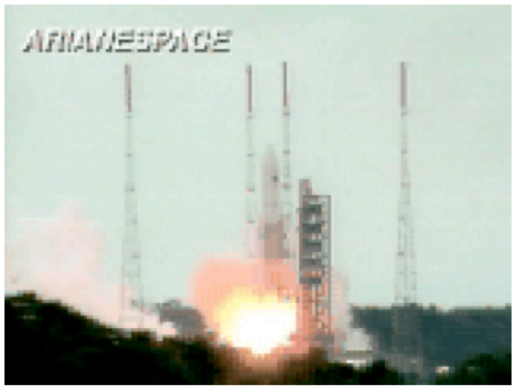
In this course, you will learn modern methods for improving software quality in a broad sense, encompassing reliability, security and performance.

This will enable you to become a better and more productive software developer, as the aspects that we will address in this course, such as software testing and debugging, comprise over 50% of the cost of software development.

You will also be able to implement these methods in specialized tools for software diagnosis and testing tasks. An example task is systematically testing an Android application in various end-user scenarios.

But let’s face it: you’re really here for the war stories.

The Ariane Rocket Disaster (1996)



Lecture under Video

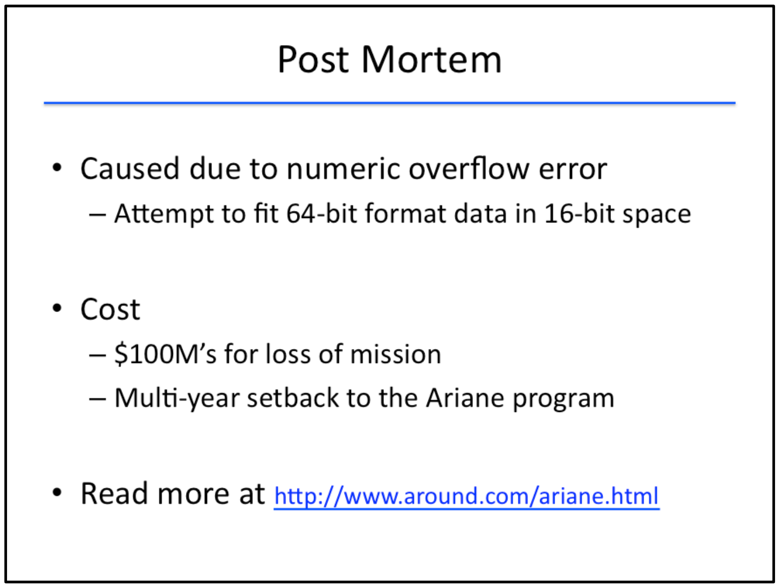
The Ariane Rocket Disaster of 1996 is a war story of epic proportions.

Here is a video of the maiden launch of the Ariane Rocket in 1996 by the European Space Agency.

Video at https://youtu.be/PK_yguLapgA?t=50s

Roughly 40 seconds after the launch, the rocket reaches an altitude of two and a half miles. But then it abruptly changes course and triggers a self-destruct mechanism, destroying its payload of expensive scientific satellites.

So why did this happen, and what was the aftermath of this disaster? Let’s take a look.



Post Mortem

- Caused due to numeric overflow error
  - Attempt to fit 64-bit format data in 16-bit space
- Cost
  - $100M's for loss of mission
  - Multi-year setback to the Ariane program
- Read more at http://www.around.com/ariane.html

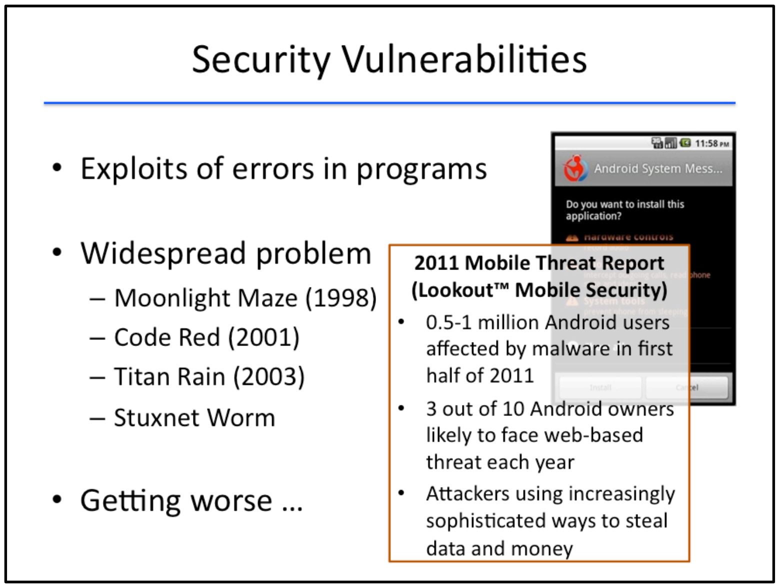
The cause of the disaster was diagnosed to be a kind of programming error called a numeric overflow error, in a program running on the Ariane rocket's onboard computer.

The error resulted from an attempt during takeoff to convert one piece of data -- the sideways velocity of the rocket -- from a 64-bit format to a 16-bit format. The number was too big to fit and resulted in an overflow error. This error was misinterpreted by the rocket's onboard computer as a signal to change the course of the rocket.

This failure translated into millions of dollars in lost assets and several years of setbacks for the Ariane Program. The methods that we will learn in this course could have prevented this error.

To read more about this disaster access the link provided in the instructor notes. [http://www.around.com/ariane.html]

Now let's look at another problem that is more earthly and affects everyday users of software.


11:58 PM
Android System Mess...
Do you want to install this application?
Hardware controls

Security Vulnerabilities

- Exploits of errors in programs
- Widespread problem
  - Moonlight Maze (1998)
  - Code Red (2001)
  - Titan Rain (2003)
  - Stuxnet Worm
- Getting worse …

2011 Mobile Threat Report (Lookout™ Mobile Security)

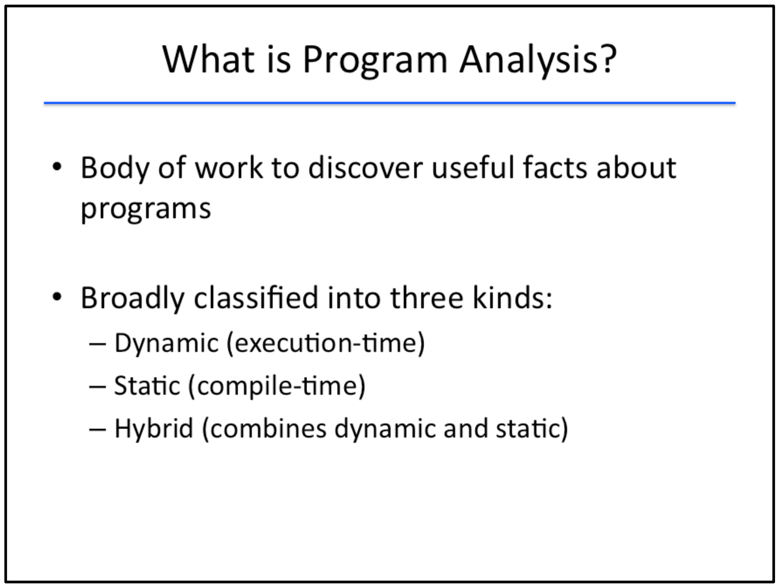
- 0.5-1 million Android users affected by malware in first half of 2011
- 3 out of 10 Android owners likely to face web-based threat each year
- Attackers using increasingly sophisticated ways to steal data and money

While the Ariane disaster was a consequence of a programming error, at least the damage was an unintended consequence. On the other hand, malicious hackers can exploit these errors in everyday mobile and web applications to compromise the security of the underlying systems and data.

This is a widespread problem, and it has been since the early days of the Internet. Several examples of programming bugs leading to security vulnerabilities you may have heard of include:

- Moonlight Maze, which probed American computer systems for at least two years since 1998,
- Code Red, which affected hundreds of thousands of Microsoft web servers in 2001,
- Titan Rain, a series of coordinated attacks on American computer systems for three years since 2003,
- and most recently Stuxnet, a computer worm that shut down Iranian nuclear facilities in 2010.

And the problem has only gotten worse with the advent of smartphones; now you too can make yourself vulnerable to programming disasters simply by installing an app. [Picture of smartphone “do you want to install” message pops up]


What is Program Analysis?

- Body of work to discover useful facts about programs
- Broadly classified into three kinds:
  - Dynamic (execution-time)
  - Static (compile-time)
  - Hybrid (combines dynamic and static)

Program analysis is the process of automatically discovering useful facts about programs. An example of a useful fact is a programming error. We saw an example of a programming error that was responsible for the Ariane disaster, and others that underlie security vulnerabilities.

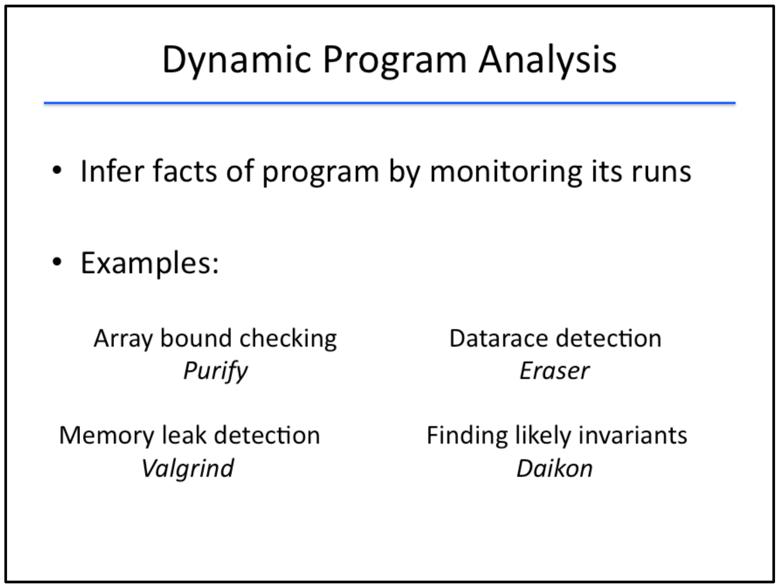
Program analysis as a whole can be broadly classified into three kinds of analyses: dynamic, static, and hybrid.

Dynamic analysis is the class of run-time analyses. These analyses discover information by running the program and observing its behavior.

Static analysis is the class of compile-time analyses. These analyses discover information by inspecting the source code or binary code of the program.

Hybrid analyses combine aspects of both dynamic and static analyses, by combining runtime and compile-time information in interesting ways.

Let's take a closer look at dynamic and static analyses.



Dynamic Program Analysis

- Infer facts of program by monitoring its runs
- Examples:

Array bound checking *Purify*	Datarace detection *Eraser*
Memory leak detection *Valgrind*	Finding likely invariants *Daikon*

Dynamic program analysis infers facts about a program by monitoring its runs.

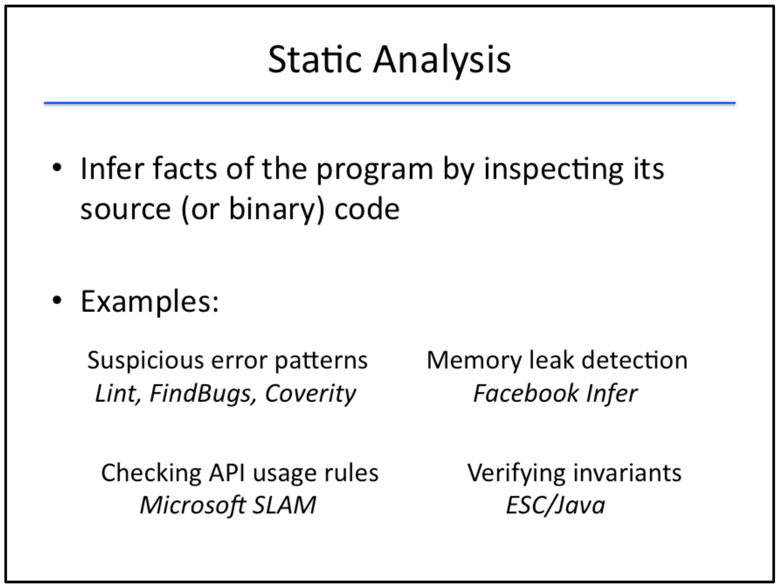
Here are four examples of well-known dynamic analysis tools.

Purify is a dynamic analysis tool for checking memory accesses, such as array bounds, in C and C++ programs.

Valgrind is a dynamic analysis tool for detecting memory leaks in x86 binary programs. A memory leak occurs when a program fails to release memory that it no longer needs.

Eraser is a dynamic analysis tool for detecting data races in concurrent programs. A data race is a condition in which two threads in a concurrent program attempt to simultaneously access the same memory location, and at least one of those accesses is a write. Data races typically indicate programming errors, as the order in which the accesses in a data race occur can produce different results from run to run.

Finally, Daikon is a dynamic analysis tool for finding likely invariants. An invariant is a program fact that is true in every run of the program.



Static Analysis

- Infer facts of the program by inspecting its source (or binary) code
- Examples:

Suspicious error patterns *Lint, FindBugs, Coverity*	Memory leak detection *Facebook Infer*
Checking API usage rules *Microsoft SLAM*	Verifying invariants *ESC/Java*

Static program analysis infers facts about a program by inspecting its code.

Here are four examples of well-known static analysis tools.

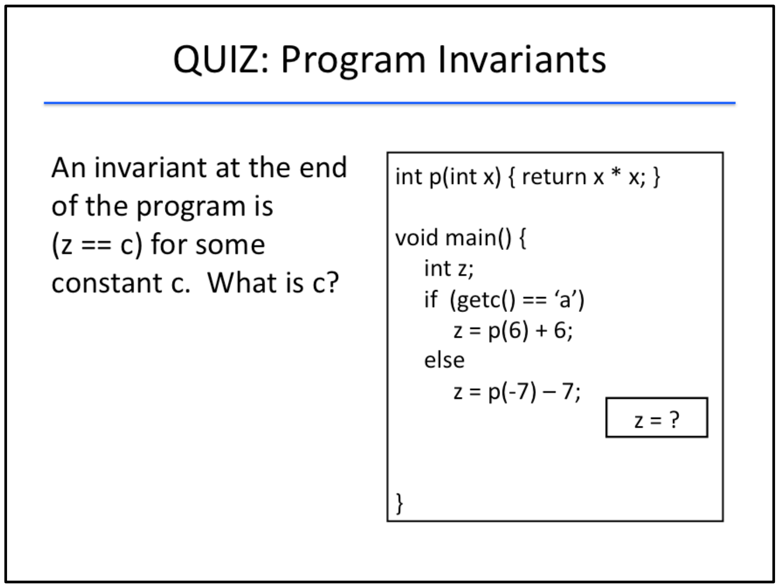
Tools such as Lint, FindBugs, and Coverity inspect the source code of C++ or Java programs for suspicious error patterns.

SLAM is a tool from Microsoft that checks whether C programs respect API usage rules. This tool is used by Windows developers to check whether device drivers use the API of the Windows kernel correctly.

Facebook Infer is a more recent static analysis tool developed by Facebook for detecting memory leaks in Android applications.

Finally, ESC/Java is a tool for specifying and verifying invariants in Java programs.

We will look at an example of an invariant next.



QUIZ: Program Invariants

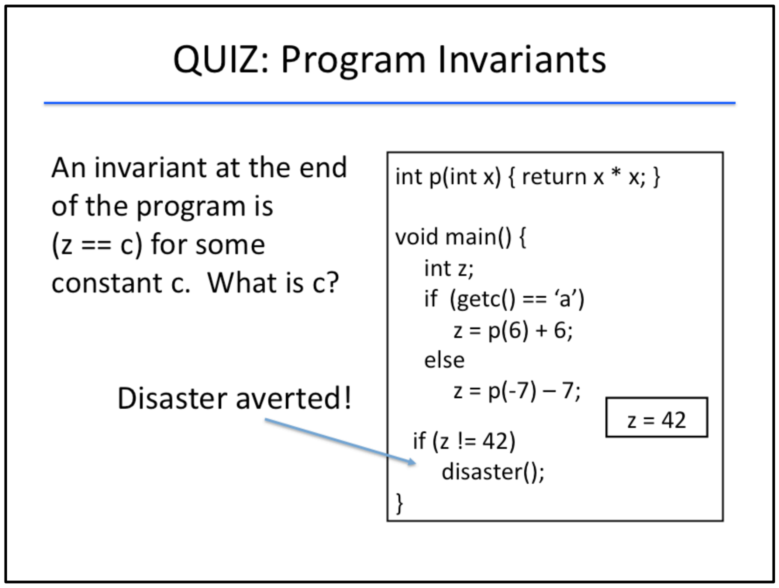
An invariant at the end of the program is (z == c) for some constant c. What is c?

```
int p(int x) { return x * x; }

void main() {
    int z;
    if  (getc() == 'a')
        z = p(6) + 6;
    else
        z = p(-7) – 7;
                              z = ?


}
```

{QUIZ SLIDE}

Let's do the following exercise to illustrate a concrete example of a useful program fact, namely, a program invariant.

Consider the following program which reads a character from the input using function getc(). If the input is the character 'a', it takes the true branch, otherwise it takes the false branch. Recall that an invariant is a program fact that is true in every run of the program. An invariant at the end of this example program is (z == c) for some constant c. What is c? Take your time and enter the value of c in this box.



QUIZ: Program Invariants

An invariant at the end of the program is (z == c) for some constant c. What is c?

```
int p(int x) { return x * x; }

void main() {
    int z;
    if  (getc() == 'a')
        z = p(6) + 6;
    else
        z = p(-7) – 7;
   if (z != 42)
       disaster();
}
```

z = 42
Disaster averted!

{SOLUTION SLIDE}

The value of c is 42. To see why, we need to reason about only two cases over all runs of this program.

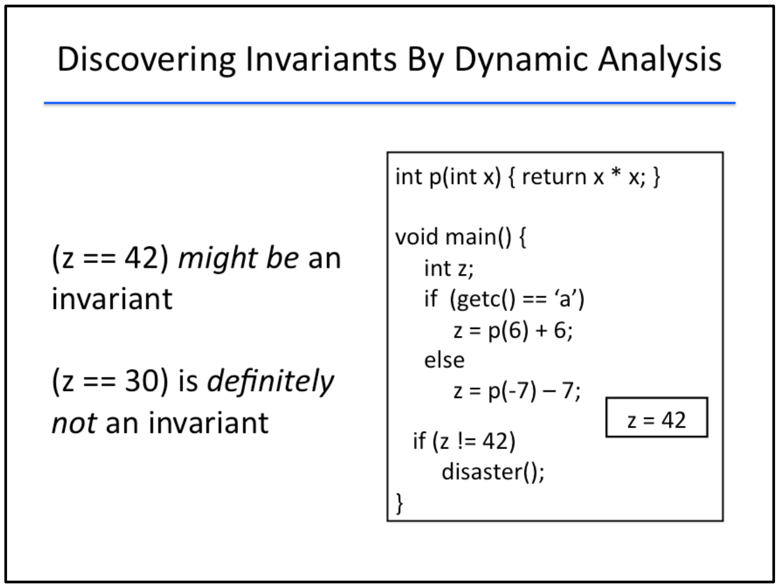
In the runs where the true branch is taken, the value of z is p(6) + 6, which is 6*6 + 6, which is 36 + 6, which is 42. *[Highlight line z = p(6) + 6]*

In the runs where the false branch is taken, the value of z is p(-7) - 7, which is (-7*-7) - 7, which is 49 - 7, which is 42 again. *[Highlight line z p(-7) - 7]*

Thus, the value of c is 42. We have thus shown that (z == 42) is a program invariant at the exit of this program.

Now let us slightly change this program to call disaster whenever the value of z is not equal to 42.

Then, notice that the invariant we just discovered is a useful fact for proving that this program can never call disaster!



Discovering Invariants By Dynamic Analysis

(z == 42) *might be* an invariant

(z == 30) is *definitely not* an invariant

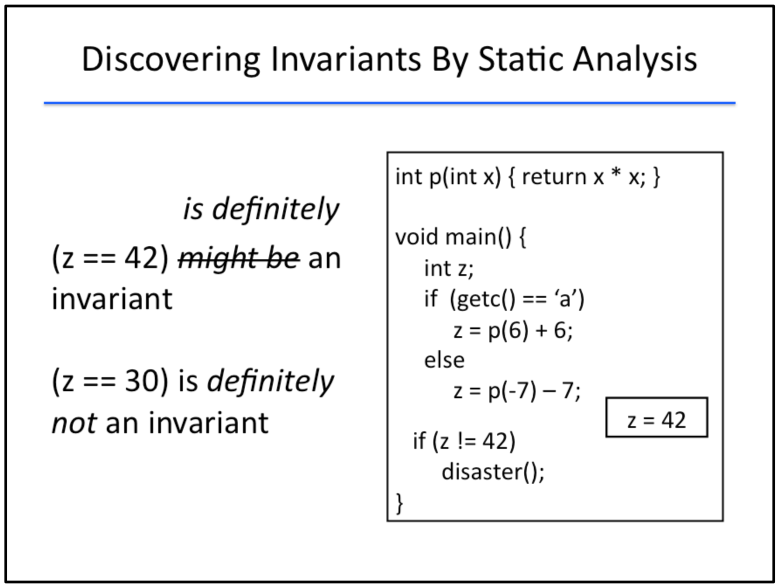
```
int p(int x) { return x * x; }

void main() {
    int z;
    if  (getc() == 'a')
        z = p(6) + 6;
    else
        z = p(-7) – 7;
   if (z != 42)
       disaster();
}
```

Now let’s see how the different kinds of program analyses fare at discovering program invariants.

Let’s first consider dynamic analysis. For simplicity, the shown program has only two paths. But in general, programs have loops or recursion, which can lead to arbitrarily many paths. Since dynamic analysis discovers information by running the program a finite number of times, it cannot in general discover information that requires observing an unbounded number of paths. As a result, a dynamic analysis tool like Daikon can at best detect *likely* invariants. From any run of the shown program, Diakon can at best conclude that (z == 42) is a *likely* invariant. It cannot prove that z will always be 42, and that the call to disaster can never happen.

This is not to say that dynamic analysis is useless. For one, the information that z might be 42 could be a useful fact. More importantly, Daikon can conclusively rule out entire classes of invariants even by observing a single run. For instance, from any run of this example program, Daikon can conclude that (z == c) is definitely not an invariant for any c other than 42.

On the other hand, to conclusively determine that (z == 42) is an invariant, and therefore showing that the program will never call disaster, we need static analysis.



Discovering Invariants By Static Analysis

is definitely
(z == 42) ~~*might be*~~ an invariant

(z == 30) is *definitely not* an invariant

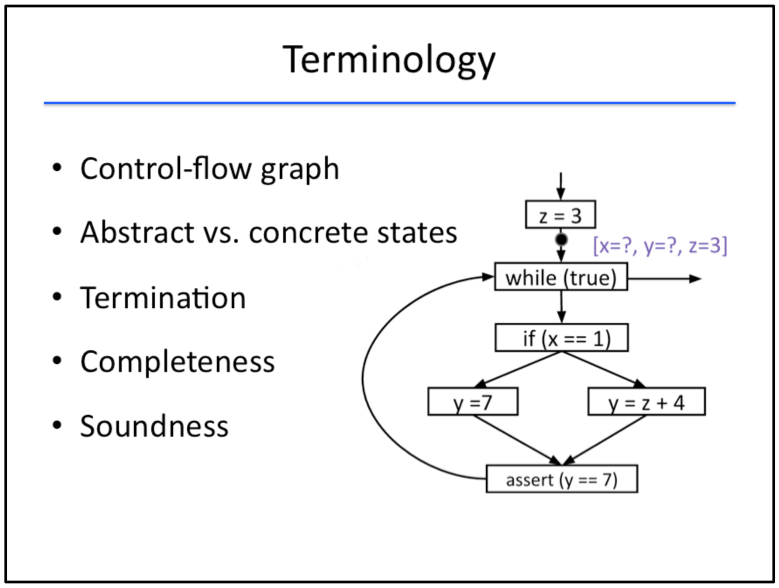
```
int p(int x) { return x * x; }

void main() {
    int z;
    if  (getc() == 'a')
        z = p(6) + 6;
    else
        z = p(-7) – 7;
   if (z != 42)
       disaster();
}
```

Now let's consider how static analysis works on this example.

Static analysis can conclusively say that (z == 42) is an invariant by inspecting the source code of the program. The reasoning it applies is similar to what we ourselves used in the quiz. Recall that we too inspected the source code to determine that the constant c has value 42.

Static analysis can therefore show at compile-time that the program will never call disaster at run-time.

You should now be able to see how the Ariane disaster could have been averted using static analysis.



Terminology

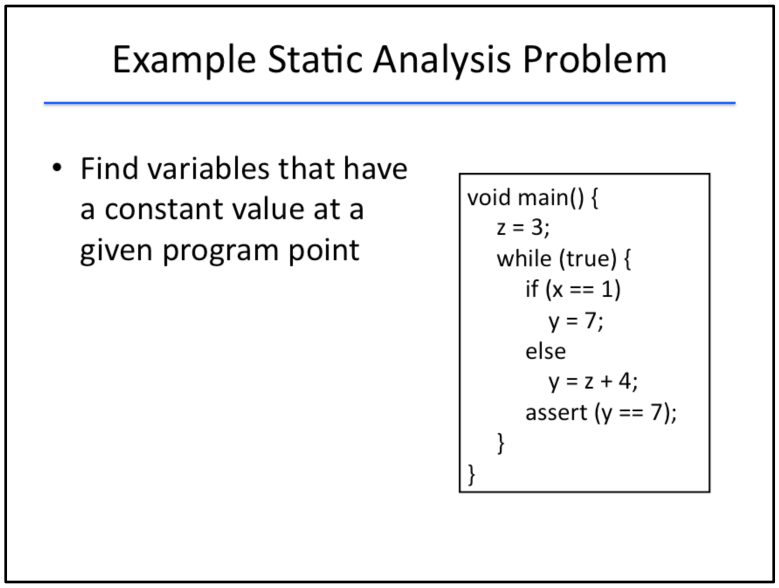
- Control-flow graph
- Abstract vs. concrete states
- Termination
- Completeness
- Soundness

Let's first introduce common terminology.

Static analysis typically operates on a suitable intermediate representation of the program. One such representation shown here is a control-flow graph. It is a graph that summarizes the flow of control in all possible runs of the program. Each node in the graph corresponds to a unique statement in the program, and each edge outgoing from a node denotes a possible successor of that node in some execution.

To achieve its stated goal, our static analysis tracks the constant values of the three variables in this program, x, y, and z, at each program point. This is called an abstract state, in contrast to a concrete state which tracks the actual values in a particular run. Since static analysis does not run the program, it does not operate directly over concrete states. Instead, it operates over abstract states, each of which summarizes a set of concrete states.

As a result of this summarization, the static analysis may fail to accurately represent the value of a variable in an abstract state, which we denote using a question mark. While this ensures the termination of the static analysis even for programs with an unbounded number of paths, it can also lead the static analysis to miss variables that have a constant value. For this reason, we say that the static analysis sacrifices completeness. Conversely, whenever the analysis concludes that a variable has a constant value, this conclusion is indeed correct in all runs of the program. For this reason, we say that the static analysis is sound.



Example Static Analysis Problem

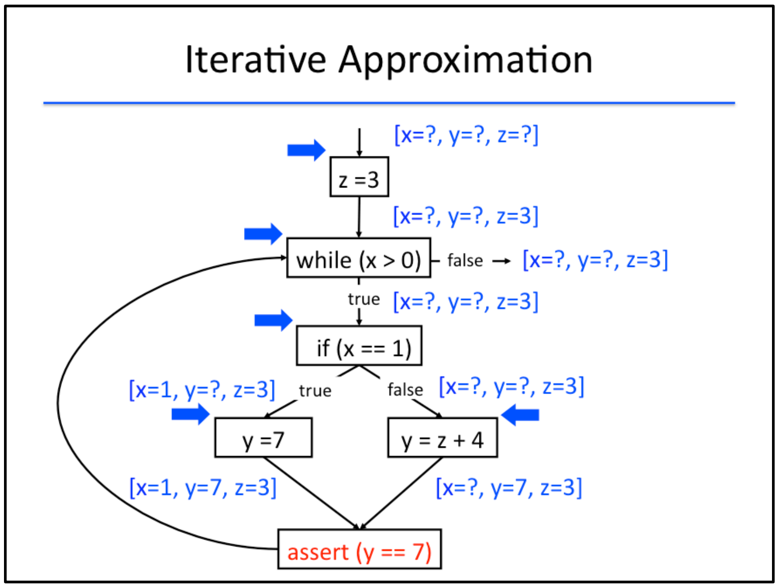
- Find variables that have a constant value at a given program point

```
void main() {
    z = 3;
    while (true) {
        if (x == 1)
            y = 7;
        else
            y = z + 4;
        assert (y == 7);
    }
}
```

Now let's examine how static analysis discovers invariants of the form (z == 42), even for programs that have an unbounded number of paths.

We can pose this question in terms of a classic static analysis problem. This problem aims to find variables that have a constant value at a given program point.

Consider the following example program which contains a loop. We will explain step-by-step how a static analysis discovers that variable y has the constant value 7 at the exit of this program.


Iterative Approximation

We use a common static analysis method called iterative approximation.

The analysis begins with unknown values of the three variables at the entry of the program. *([x=?, y=?, z=?] fades in)* At each step, the analysis updates its knowledge about the values of the three variables at each program point. The analysis does this update based upon the information that it has inferred at the immediate predecessors of that program point.

For instance, after the statement that assigns 3 to z, *(the first [x=?, y=?, z=3] fades in)* the analysis knows that the value of z is the constant 3.

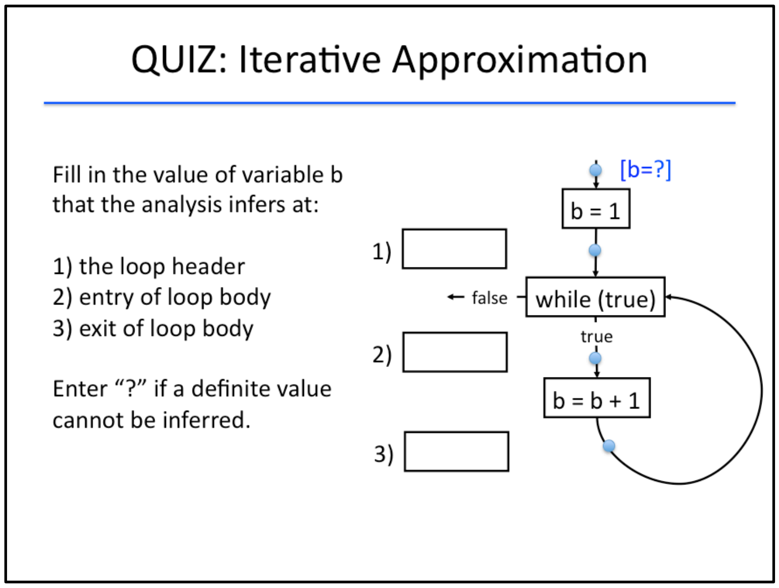
Another interesting update occurs in the true branch of the condition that checks whether the value of x is 1. *([x=1, y=?, z=3] fades in)* After taking this branch, the analysis knows that the value of x must be the constant 1. Notice that in the false branch of this condition, the analysis does not know whether x has a constant value in all runs of this program. *([x=?, y=?, z=3] next to y=z+4 fades in)* So it continues to indicate that the value of x is unknown.

Similarly, after the statement that assigns 7 to y, the analysis knows that the value of y is the constant 7. *([x=1,y=7,z=3] fades in)* The analysis has thus concluded that, every time this program point is reached in any run, x has value 1, y has value 7, and z has value 3.

Now let's look at the statement that assigns the expression z + 4 to y. Since the analysis had previously discovered that the value of z before this statement is the constant 3, it can conclude that the value of y after this statement must be 3 + 4, which is 7. *([x=?, y=7, z=3] fades in)*

At this point, the analysis has concluded that, at each immediate predecessor of the assertion, the value of y is 7. It thereby concludes that the value of y in the assertion must be 7, and therefore that the assertion is valid.

The term iterative approximation implies that in general, the analysis might need to visit the same program point multiple times. This is because of the presence of loops, which can require the analysis to update facts that were previously inferred by the analysis at the same program point. We will emphasize this aspect in the following quiz.



QUIZ: Iterative Approximation

Fill in the value of variable b that the analysis infers at:

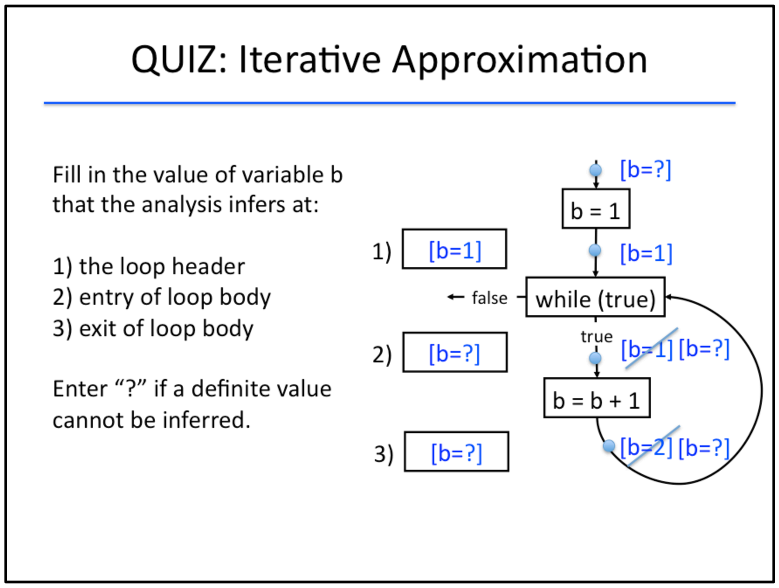
1) the loop header
2) entry of loop body
3) exit of loop body

Enter "?" if a definite value cannot be inferred.

{QUIZ SLIDE}

Consider the following program. The analysis begins with an unknown value for variable b at the start of this program. In each of the three boxes shown, fill in the value of variable b that the analysis infers at the corresponding program point after completing its analysis. We will call these program points the loop header, the entry of the loop body, and the exit of the loop body.

If the analysis cannot infer a definite value for b, enter a question mark into the box.



QUIZ: Iterative Approximation

Fill in the value of variable b that the analysis infers at:

1) the loop header
2) entry of loop body
3) exit of loop body

Enter "?" if a definite value cannot be inferred.

1) [b=1]

2) [b=?]

3) [b=?]

[b=?]
b = 1
[b=1]
false
while (true)
true
~~[b=1]~~ [b=?]
b = b + 1
~~[b=2]~~ [b=?]

{SOLUTION SLIDE}

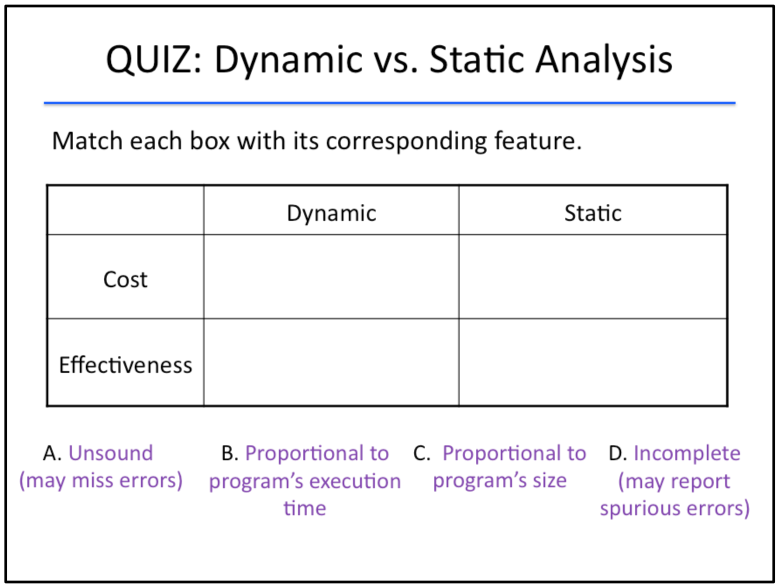
The value of b in the first box is 1. This is because immediately after the assignment of 1 to b, our static analysis knows that the value of b is 1.

As the analysis proceeds, it discovers that the value of b at the entry of the loop body is still 1.

Similarly, it discovers that the value of b at the exit of the loop body is 2. But the analysis is not done yet. It must analyze the loop again to ensure that these values are indeed sound.

The analysis revisits the entry of the loop body. This time, it notices that the value of b can be 1 or 2. So it updates the value of b at the entry of the loop body to unknown. Continuing further, the analysis updates the value of b at the exit of the loop body to unknown as well.

Due to these updates, the analysis analyzes the loop yet again. But this time, it concludes that the values of b at the entry and exit of the loop body have saturated. Therefore, the correct value of b in the 2nd and 3rd boxes is the unknown value.



QUIZ: Dynamic vs. Static Analysis

Match each box with its corresponding feature.

	Dynamic	Static
Cost		
Effectiveness		

A. Unsound (may miss errors)

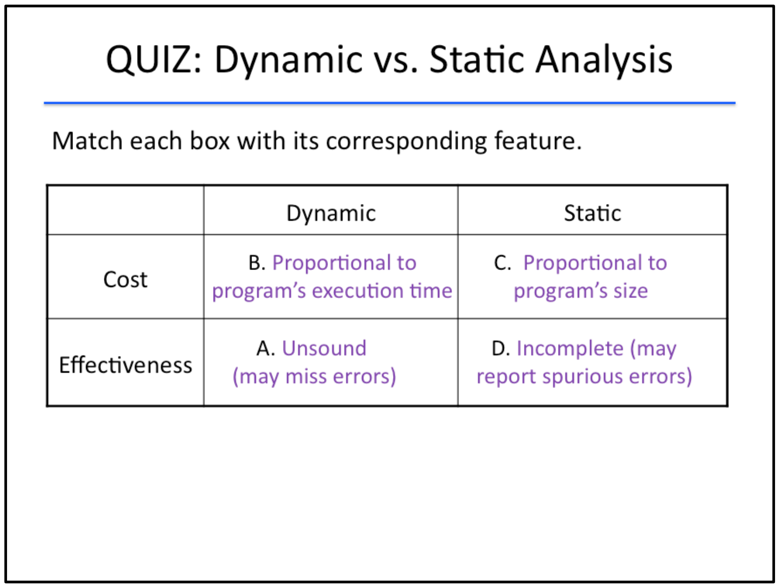
B. Proportional to program's execution time

C. Proportional to program's size

D. Incomplete (may report spurious errors)

{QUIZ SLIDE}

OK, time for another quiz. Dynamic and static analyses strike different tradeoffs in terms of their cost and effectiveness. Match each box with its corresponding feature.



QUIZ: Dynamic vs. Static Analysis

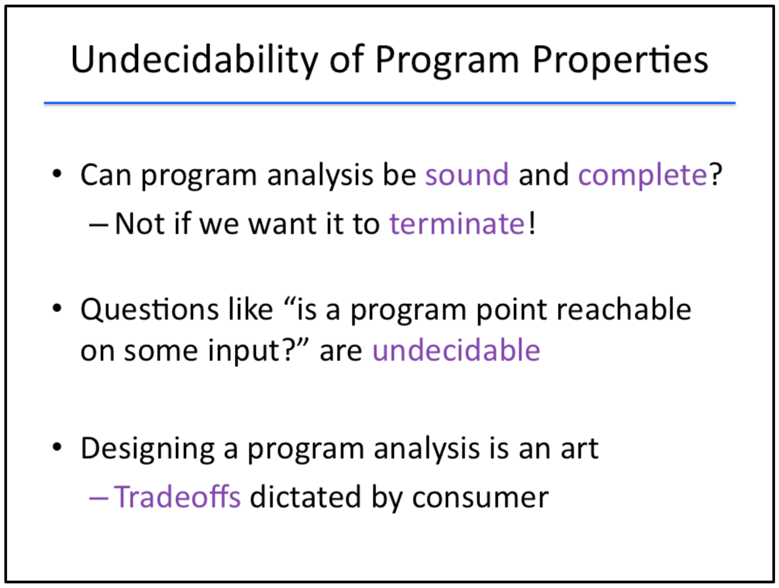
Match each box with its corresponding feature.

	Dynamic	Static
Cost	B. Proportional to program's execution time	C. Proportional to program's size
Effectiveness	A. Unsound (may miss errors)	D. Incomplete (may report spurious errors)

{SOLUTION SLIDE}

Let's review the answers. First we will focus on cost. Since dynamic analysis gathers information by running the program, its cost is proportional to the execution time of the program. A longer run thus costs more than a shorter one. Static analysis, on the other hand, gathers information by inspecting the program's code, and therefore its cost is proportional to the size of the program's source code. A larger program thus costs more than a smaller one.

Now let's look at effectiveness. As we saw in the example about program invariants, a dynamic analysis may miss errors, as it inspects only a finite number of runs whereas the program may contain an unbounded number of paths, some of which are not covered by those runs. We say that a dynamic analysis is "unsound": in other words, it may produce false negatives. Static analysis, on the other hand, does not miss errors but it may report spurious issues. We say that a static analysis is "incomplete": in other words, it may produce false positives.



Undecidability of Program Properties

- Can program analysis be sound and complete?
  - Not if we want it to terminate!
- Questions like "is a program point reachable on some input?" are undecidable
- Designing a program analysis is an art
  - Tradeoffs dictated by consumer

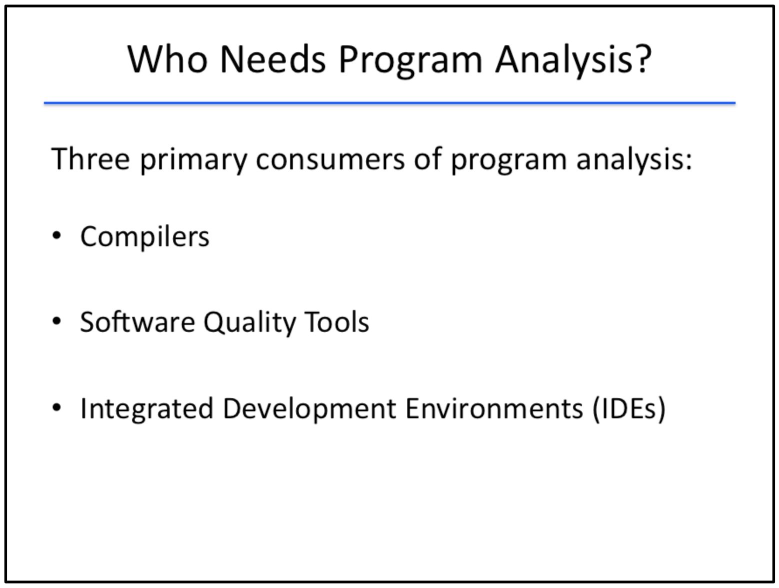
You might be wondering whether it is possible for a program analysis to guarantee both soundness and completeness: no false positives nor false negatives. The answer is: not if we want the analysis to eventually finish!

Even seemingly simple program properties for realistic programming languages like C and Java are undecidable. An example such property is whether a given point in a given program is reachable on some input to that program.

You can find a link to recommended reading on the topic of undecidability in the instructor notes on this page.

https://en.wikipedia.org/wiki/Undecidable_problem

Designing a program analysis is thus an art that involves striking a suitable tradeoff between termination, soundness, and completeness. This tradeoff is typically dictated by the consumer of the program analysis. Let's look at the primary consumers of program analysis next.

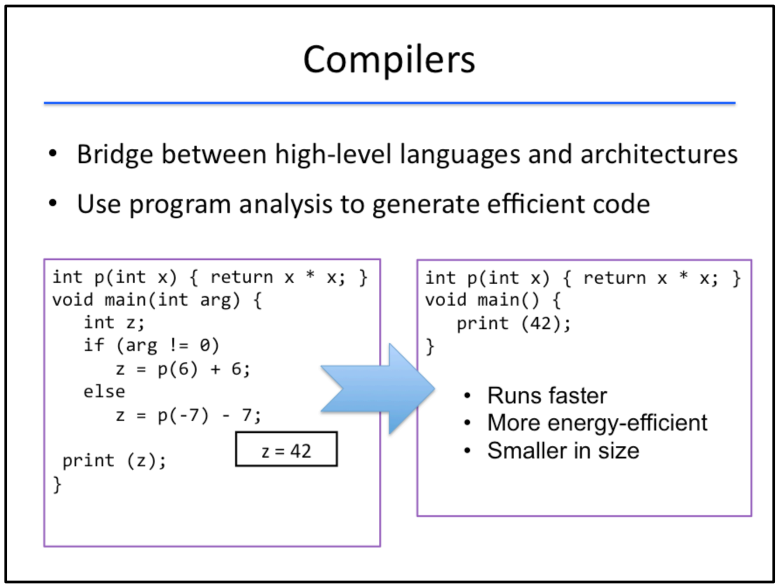


Who Needs Program Analysis?

Three primary consumers of program analysis:

- Compilers
- Software Quality Tools
- Integrated Development Environments (IDEs)

There are three primary consumers of program analysis: compilers, software quality tools, and integrated development environments.



Compilers

- Bridge between high-level languages and architectures
- Use program analysis to generate efficient code

```
int p(int x) { return x * x; }
void main(int arg) {
   int z;
   if (arg != 0)
      z = p(6) + 6;
   else
      z = p(-7) - 7;

 print (z);
}
```

z = 42

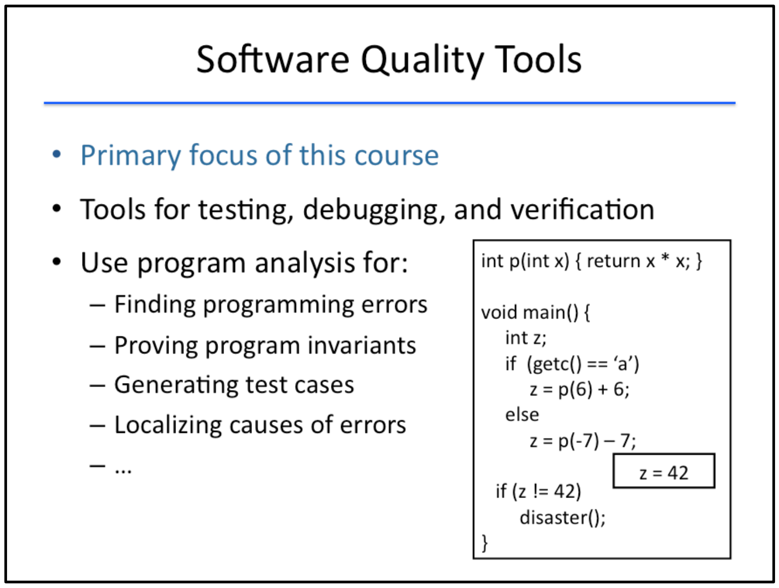
```
int p(int x) { return x * x; }
void main() {
   print (42);
}
```

- Runs faster
- More energy-efficient
- Smaller in size

Compilers bridge the gap between high-level programming languages and advanced computer architectures. They use program analyses to generate efficient code on a target architecture for programs written in a high-level source language.

Let us see a simple example of how a program analysis can help a compiler generate more efficient code. Consider this example program.

We saw earlier in this lesson how a static analysis can discover the program invariant (z == 42) at the end of this program. A compiler can use this invariant to simplify this program. The simplified program simply prints 42. It is easy to see that this simplified program is more efficient than the original program: it runs faster, it is more energy-efficient, and it is smaller in size.



Software Quality Tools

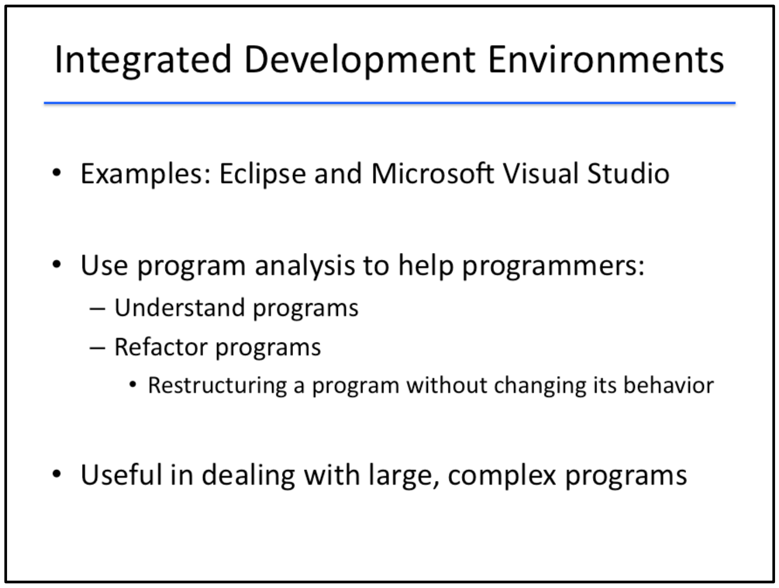
- Primary focus of this course
- Tools for testing, debugging, and verification
- Use program analysis for:
  - Finding programming errors
  - Proving program invariants
  - Generating test cases
  - Localizing causes of errors
  - …

```
int p(int x) { return x * x; }

void main() {
   int z;
   if  (getc() == 'a')
      z = p(6) + 6;
   else
      z = p(-7) – 7;
                              z = 42
  if (z != 42)
     disaster();
}
```

The second key consumer of program analysis is software quality tools, which will be the primary focus of this course.

This category broadly includes tools programmers use for tasks to improve software quality, such as testing, debugging, and verification. These tools use program analyses for various purposes such as finding programming errors, proving program invariants, generating test cases, and localizing the causes of errors.

Consider this example program again. The invariant (z == 42) discovered at this program point by a static analysis could be used by a program verification tool to prove that this program will never call disaster.



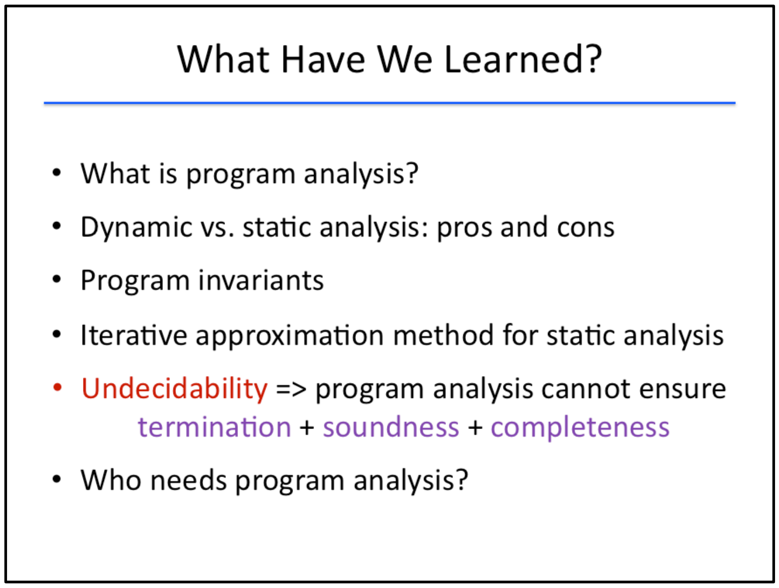
Integrated Development Environments

- Examples: Eclipse and Microsoft Visual Studio
- Use program analysis to help programmers:
  - Understand programs
  - Refactor programs
    - Restructuring a program without changing its behavior
- Useful in dealing with large, complex programs

The third main consumer of program analysis is integrated development environments such as Eclipse or Microsoft Visual Studio.

Such environments use program analyses to help programmers understand programs and refactor programs, which is the process of restructuring a program without changing its external behavior.

These features are especially needed when dealing with large, complex programs which are common in practice.


What Have We Learned?

- What is program analysis?
- Dynamic vs. static analysis: pros and cons
- Program invariants
- Iterative approximation method for static analysis
- Undecidability => program analysis cannot ensure termination + soundness + completeness
- Who needs program analysis?

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

First, we introduced program analysis, a process for automatically discovering useful facts about programs.

We then discussed two kinds of program analyses: dynamic and static analysis. The primary difference between these two kinds of analyses is that dynamic analysis works by running the program whereas static analysis works by inspecting the program's code. We discussed the pros and cons of these two kinds of analyses.

We learnt about program invariants and their role as useful program facts. We discussed how dynamic analysis can discover likely invariants, and how static analysis can prove invariants.

We also saw step-by-step how static analysis can prove a certain kind of program invariant using the method of iterative approximation.

We learnt that the undecidability of even simple program properties prevents program analyses from simultaneously guaranteeing the three desirable features of termination, soundness, and completeness.

Finally, we discussed the three main consumers of program analysis: compilers, software quality tools, and integrated development environments.

In the next lesson, we will learn about dataflow analysis, a popular approach to static analysis that embodies the method of iterative approximation.