Hack Your Language!

CSE401 Winter 2016

Introduction to Compiler Construction

Ras Bodik
Alvin Cheung
Maaz Ahmad
Talia Ringer
Ben Tebbs

Lecture 13: Static typing

Why types?
Bitflip attack

Announcements

- Midterm grades released
  - Submit requests on gradescope within the next week
  - We will regrade the **entire** exam
- PA3 due Tuesday at 11pm
- Project milestones released
  - We will post comments to your proposals by this weekend
  - Two part assignment with different due dates

Outline for today

Motivation for types:

- Language-based security
- Runtime program performance

Introduction to types

- Static vs. dynamic types
- The “bitflip” exploit

Static vs. dynamic; value vs. variable

Dynamic = known at run time

- specific to a given program input

Static = known at compile time

- not specific to given input => must be true for all inputs

Type = set of values and operations on them

- example: ints are -2^{32}, …, 0, … $2^{32}-1$, with operations +, -, …

Dynamic type of a variable

- is the type of the value stored in the variable at runtime

Static type of a variable

- Annotated by programmers or inferred by the compiler
- type of *all* values that the variable *might* hold at runtime

Security

Introduction to static typing, Part I

You are playing an MMO

Your browser connects to a game server

The server provides ways to:

- return player's data
- return currently running games

Server must restrict access to such data

One way to do so is via language-based security

- using private fields of objects

Private fields of objects

Private object fields

We can create an object with a private field

- the private field stores a password that can be checked against a guessed password for equality but the stored password cannot be leaked

We don't need a statically typed language for this

- This can be done even in Lua

Next slide shows the code

Object with a private field

```
// Usage of an object with private field
def safeKeeper = SafeKeeper(“401rocks”)
print safeKeeper.checkPassword(“401stinks”)  -->  False

// Implementation of an object with private field
function SafeKeeper (password)
    def pass_private = password

    def checkPassword (pass_guess) {
        pass_private == pass_guess
    }
    // return the object, which is a table
    { checkPassword = checkPassword }
}
```

Q: Why is it pass_private secure?

Let's try to read out the private field!

Assume I agree to execute any code you give me.
Can you print the password (without trying all passwords)?

```
def safeKeeper = SafeKeeper("401rocks")
def yourFun = <paste any code here>
// I am even giving you a ref to keeper
yourFun(safeKeeper)
```

This privacy works great, under certain assumptions. Which features of the 401 language do we need to disallow to prevent reading of pass_private?

1. overriding == with our own method that prints its arguments
2. access to the environment of a function and printing the content of the environment

(such access could be allowed to facilitate debugging, but it destroys privacy)

Same in Java, using private fields

```
class SafeKeeper {
    private long pass_private;
    SafeKeeper(password) { pass_private = password }

    Boolean checkPassword (long pass_guess) {
        return pass_private == pass_guess
}   }

SafeKeeper safeKeeper = new SafeKeeper(“401rocks”)
print safeKeeper.checkPassword(“401stinks”)  -->  False
```

Challenge: how to read out the private field?

Different language. Same challenge.

```
SafeKeeper safeKeeper = new SafeKeeper(19238423094820)
<paste your code here; it can refer to 'safeKeeper'>
```

Compiler rejects program that attemps to read the private field

That is, p.private_field will not compile to machine code

But some features of Java need to be disallowed to prevent reading of pass_private.

- Reflection, also known as introspection

  read about the ability to read private fields with java reflection API)

Summary of privacy with static types

It's frustrating to the attacker that

(1) she holds a pointer *a* to the Java object, and

(2) knows that password is at address *a*+16 bytes

yet she can't read out `pass_private` from that memory location.

Why can't any program read that field?

0. Compiler will reject program with p.private_field
1. Type safety prevents variables from storing incorrectly-typed values.

   B b = new A() disallowed by compiler unless A extends B
2. Array-bounds checks prevent buffer overflows
3. Can't manipulate pointers (addresses) and hence cannot change where the reference points.

Together, these checks prevent execution of arbitrary user code…

Unless the machine breaks! (will see later in lecture)

Performance

Introduction to static typing, Part II

What makes a language implementation fast?

Performance as measured on "language shootout"

An incomplete list of languages. See web site for more.

http://benchmarksgame.alioth.debian.org/

Norm performance	language + implementation (2007)
1.0	C / gcc
1.2	Pascal Free Pascal
1.5	Java 6 -server
1.6	Lisp SBCL
1.8	BASIC FreeBASIC
3.1	Fortran G95
11	Lua
14	Scheme MzScheme
17	Python
21	Perl
23	PHP
44	JavaScript SpiderMonkey
48	Ruby

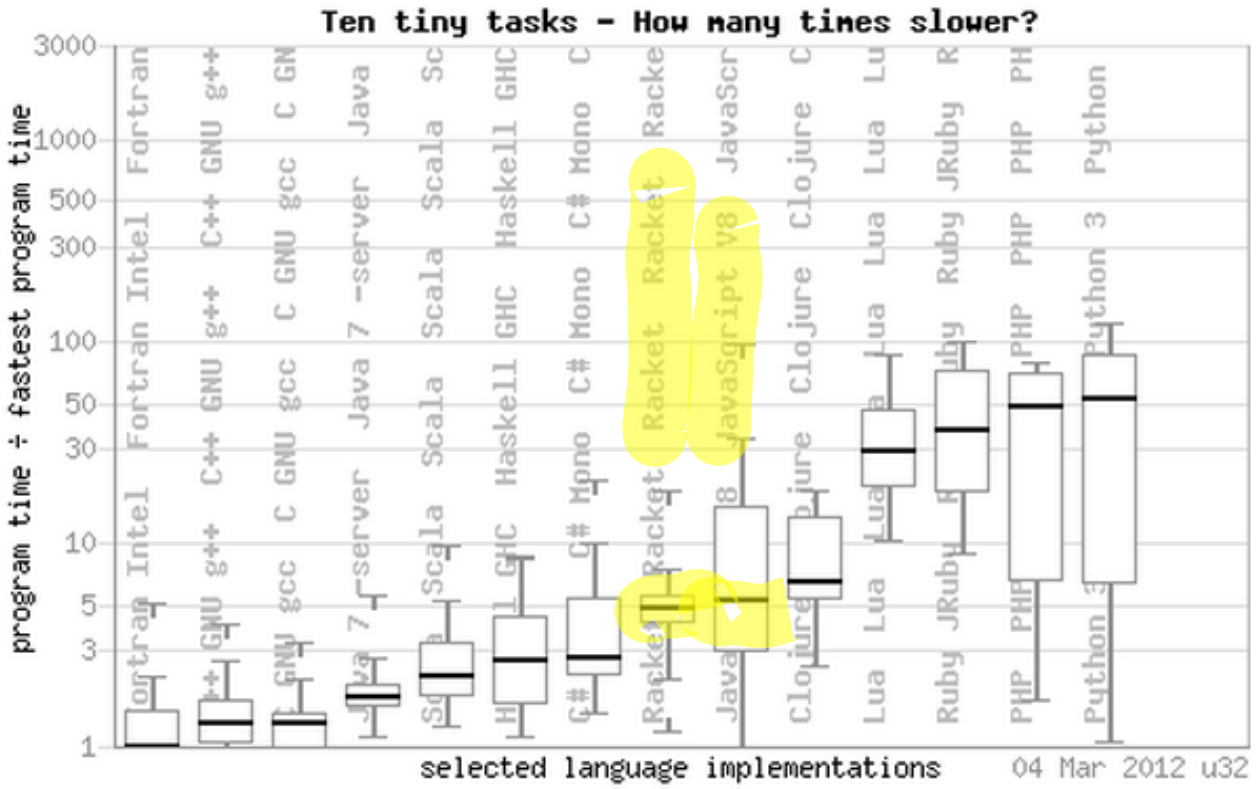
Performance diffs form equivalence classes

Notice the huge step between 3x and 10x

What might be the source of the difference?

All languages under 3x are statically typed

For comparison, more recent results (2012)



compilers for dynamically typed languages improved since 2007

- partly motivated by browser performance wars (JavaScript JITs)
- Languages such as JS and Racket use JIT to combine both static and runtime info (will revisit in 2 weeks)

2012 results

chart

compare 2	|-	|---	25%	median	75%	---|	-|
Fortran Intel	1.00	1.00	1.00	**1.00**	1.49	2.24	5.15
C++ GNU g++	1.00	1.00	1.03	**1.27**	1.68	2.65	4.06
C GNU gcc	1.00	1.00	1.00	**1.30**	1.47	2.17	3.24
ATS	1.01	1.01	1.17	**1.37**	1.60	2.24	7.95
Ada 2005 GNAT	1.01	1.01	1.35	**1.62**	2.44	4.08	7.55
Java 7 -server	1.11	1.11	1.58	**1.76**	2.02	2.68	5.62
Scala	1.24	1.24	1.81	**2.18**	3.22	5.33	9.79
Pascal Free Pascal	1.38	1.38	2.01	**2.39**	2.84	4.10	5.36
Haskell GHC	1.10	1.10	1.64	**2.64**	4.34	8.38	8.73
C# Mono	1.44	1.44	2.26	**2.72**	5.46	10.26	20.52
Clean	1.76	1.76	2.11	**3.01**	4.15	7.21	11.17
OCaml	1.55	1.55	2.02	**3.40**	4.90	6.26	6.26
Lisp SBCL	1.02	1.02	1.87	**3.81**	4.99	9.67	10.87
F# Mono	1.43	1.43	2.53	**3.97**	5.62	10.24	18.28
Racket	1.17	2.16	4.19	**4.79**	5.55	7.58	18.58
Go	1.99	1.99	2.84	**5.15**	7.57	9.48	9.48
JavaScript V8	1.00	1.00	2.97	**5.27**	15.30	33.80	98.78
Clojure	2.47	2.47	5.47	**6.40**	13.79	18.59	18.59
Erlang HiPE	3.28	3.28	7.20	**12.79**	23.16	30.80	30.80
Smalltalk VisualWorks	5.35	5.35	12.10	**14.47**	25.93	46.68	78.54
Lua	10.45	10.45	19.68	**28.51**	46.85	86.87	86.87
Ruby JRuby	9.02	9.02	18.65	**37.10**	72.73	100.02	100.02
Ruby 1.9	6.24	6.24	9.28	**43.79**	89.90	210.84	262.44
PHP	1.68	1.68	6.58	**48.33**	71.26	79.70	79.70
Python 3	1.06	1.06	6.50	**52.17**	86.22	126.24	126.24
Mozart/Oz	6.73	6.73	34.71	**57.66**	76.90	140.17	147.58
Perl	2.22	2.22	7.31	**57.75**	103.44	220.49	220.49
C CINT	71.42	71.42	179.97	**251.18**	379.29	678.27	4790.34

What do compilers know thanks to static types?

Dynamically-typed languages (Python/Lua/JS):

```
function foo(arr) {
    return arr[1]+2
}
```

Questions that compilers must solve:

- Can this code throw exceptions?
- Which +?
- How to access [1] from arr?

Statically-typed languages (Java/C#/Scala):

```
function foo(arr:int[]) : int {
    return arr[1]+2
}
```

Declared types lead to compile-time facts

Let's discuss our example: arr[1] + 2

The + operator/function:

Declared types lead to compile-time facts

Let's discuss our example: arr[1] + 2

The + operator/function:

In **Java:** we know at compile time that + is an integer addition, because type declarations tell the compiler the types of operands.

In **JS:** we know at compile time that + could be either int addition or string concatenation. Only at runtime, when we know the types of operand values, we know which of the two functions should be called.

Declared types lead to compile-time facts

Does a Python compiler know that variable arr will refer to a value of indexable type?

Declared types lead to compile-time facts

Does a Python compiler know that variable arr will refer to a value of indexable type?

It looks like it should, because arr is used in the indexing expression arr[1].

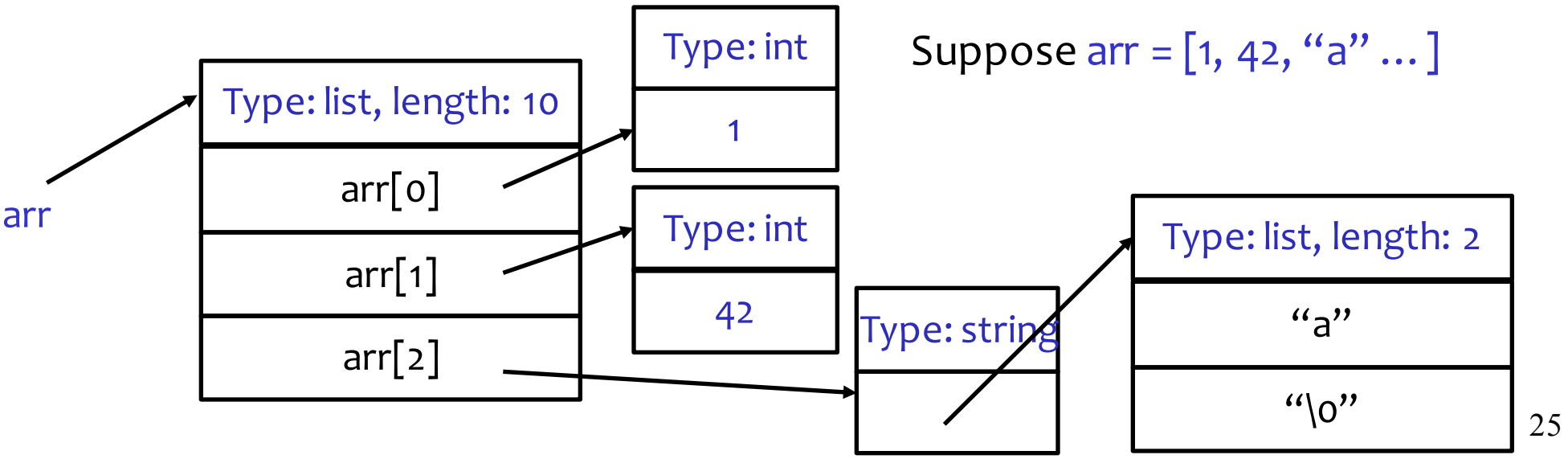
But Python does not even know this fact for sure. After all, foo could be legally called with a float argument, say foo(3.14). Yes, foo will throw an exception in this case, at arr[1], but the point is that the compiler must generate code that checks (at runtime) whether the value in arr is an indexable type. If not, it will throw an exception.

Data structure representation

In Python, `arr[i]` must check at runtime:

- is (the value of) arr an indexable object (list or a dictionary)?
- what is type (of value in) of arr[1]?

The representation of array of ints must facilitate these runtime questions:



Compare this with array of ints in Java

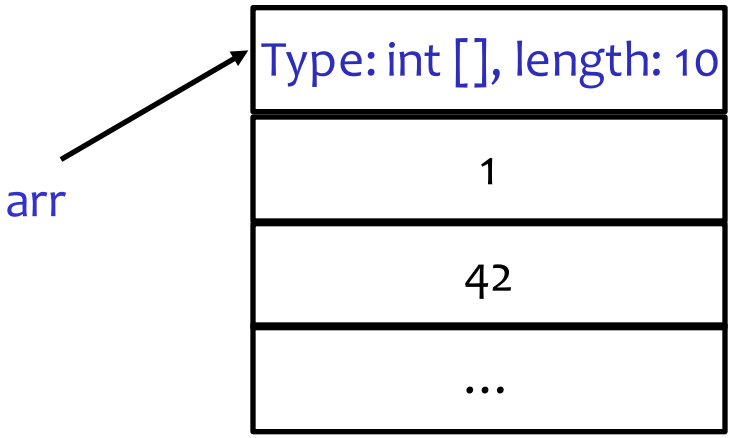
Java arrays must be homogeneous

- All elements are of the same type (or subtype)

We know these types at compile time

- So the two questions that Python asks at runtime can be skipped at Java runtime, because they are answered from static type declarations at compile time

Hence Java representation of arrays of ints can be:

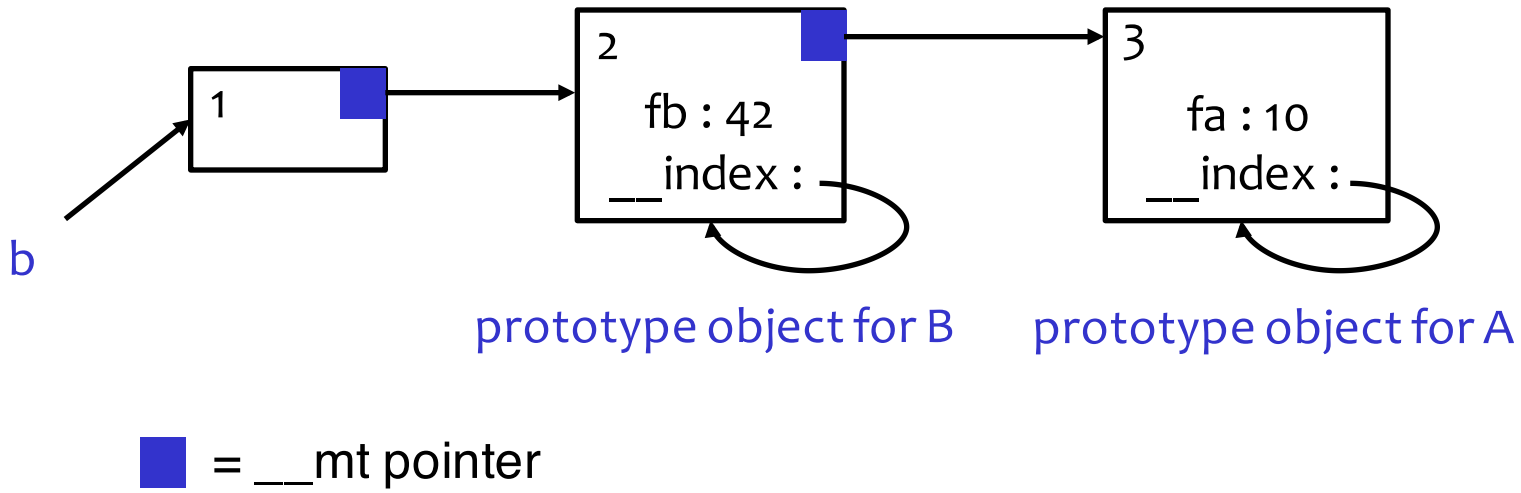


Field access in presence of inheritance

Assume Lua objects:

- Assume class A with field fa,
- subclass B of A with field fb, and
- object b is an instance of B

Count dictionary accesses when we do b.fa?



The same in Java

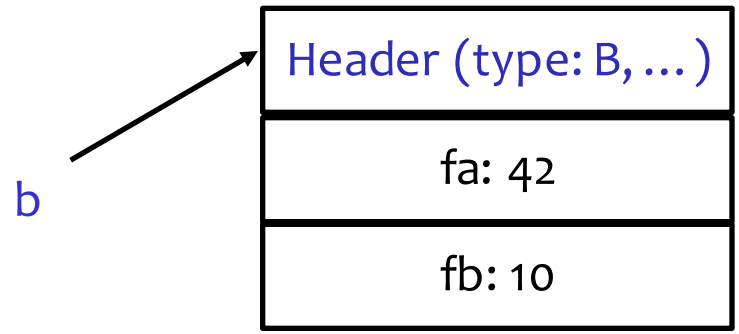
Assume the same class hierarchy:

Assume class A with field fa,

subclass B of A with field fb, and

object b is an instance of B

What is the Java object layout for b?



Compiler can now generate instructions of the form: `load(b, 16)`

Objects in (most) dynamically typed languages

Cost of access to objects built from dictionaries

Such objects are in Lua or JavaScript

Reading/writing an attribute residing in the object:

1

Reading an attribute in a prototype:

Includes inherited methods and constants (class vars):

1 + 2n (n = length of the class hierarchy)

These dictionary lookups use string-valued keys

The JIT compiler might be able to optimize them

Lesson

Programs in languages with static typing run faster.

Reason: the compiler has more information about the program and can thus generate more efficient code.

- Better layout of objects: structs rather than dictionaries
- No dynamic examination of types of values of base types such as ints, floats

Performance is one reason why we use static types.

- But JIT (runtime) compilers for dynamically typed languages can obtain some of this information at runtime, and produce improved code.

Next

Type safety provides strong security guarantees.

But certain assumptions must hold first:

- banning some constructs of the language
- integrity of the hardware platform

These are critical. Failure to provide these permits type system subversion.

Type system subversion

means and consequences

Manufacturing a Pointer in C

Attack in C language

Before we describe the attack in Java, how would one forge (manufacture) a pointer in C

```
union {  int i; char * s; } u;
```

Here, i and s are names for the same location.

u.i = 1000

u.s[0] --> reads the character at address 1000

http://stackoverflow.com/questions/4748366/can-we-use-pointer-in-union

An illustration

```
struct A { int a; }; struct B { double b; };
int main () {
  struct A * a = (struct A *)malloc(sizeof(struct A));
  a->a = 10;
  struct B * b = (struct B *)a;
  printf("b: %f\n", b->b);
}
```

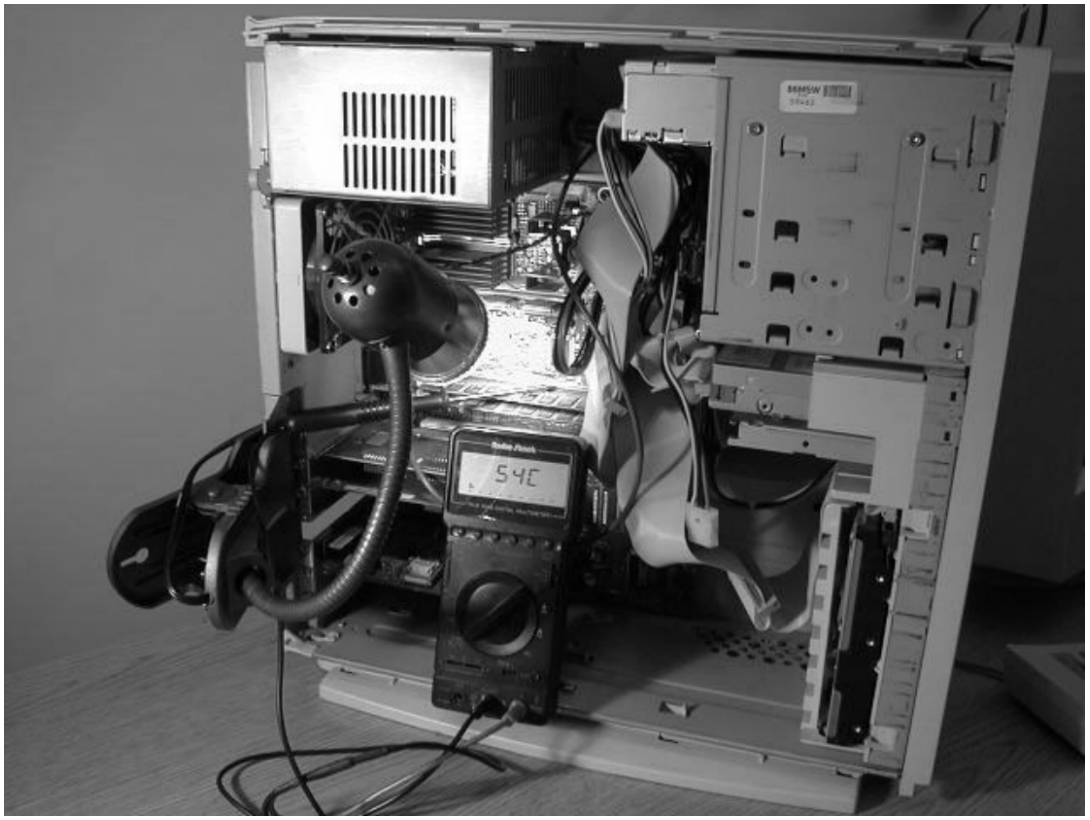
```
$ ./a.out
b: 0.000000

$ ./a.out
b: -0.000000

$ ./a.out
b: -
3105036184601424764863287854409128521424031322875637732419490432442543026
9594990429103660979270681167259778911549969386434600992581431495917188135
1099776175729828614087496746441130079273810063142804673252690491933127660
8830997266432.000000
```

How to create a hardware error?

Memory Errors

A flip of some bit in memory

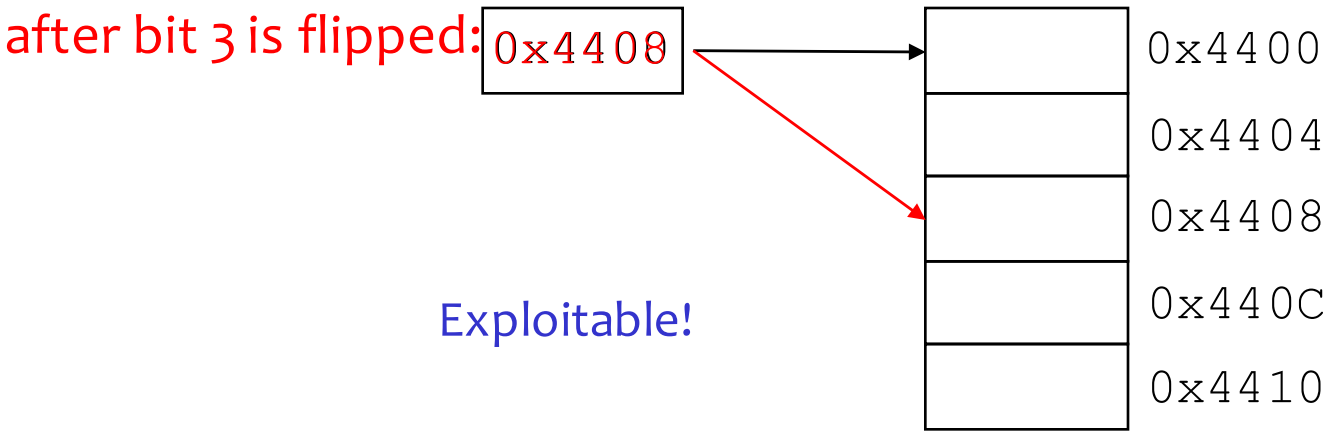
- Can be caused by cosmic ray, or deliberately through radiation (heat)



Effects of memory errors

Bitflip manufactures a pointer

except that we cannot control what pointer and in which memory location.



Manufacturing a Pointer in Java and Exploiting it

Overview of the Java Attack

Step 1: use a memory error to obtain two variables p and q, such that

1. p == q (i.e., p and q point to same memory loc) and
2. p and q have incompatible, custom static types

Cond (2) normally prevented by the Java type system.

Step 2: use p and q from Step 1 to write values into arbitrary memory addresses

- Fill a block of memory with desired machine code
- Overwrite dispatch table entry to point to block
- Do the virtual call corresponding to modified entry

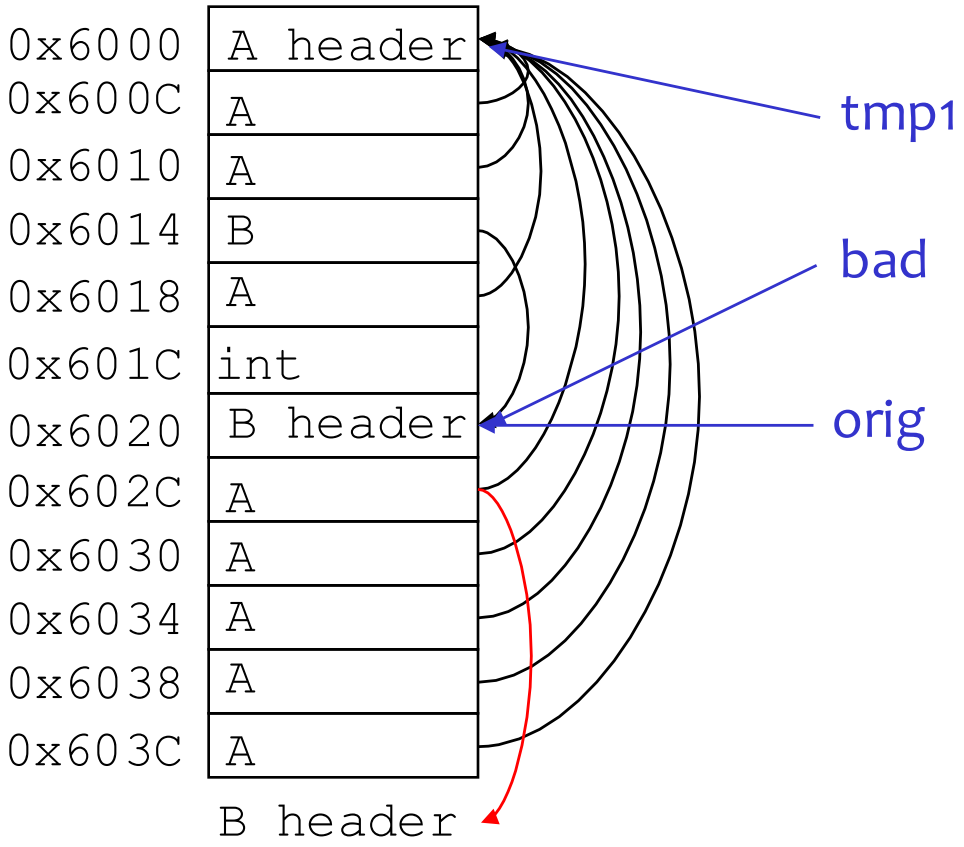
The two custom classes will form C-like union

```
class A {
  A a1;
  A a2;
  B b;    // for Step 1
  A a4;
  int i; // for address
         // in Step 2
}
```

```
class B {
  A a1;
  A a2;
  A a3;
  A a4;
  A a5;
}
```

Assume 3-word object header

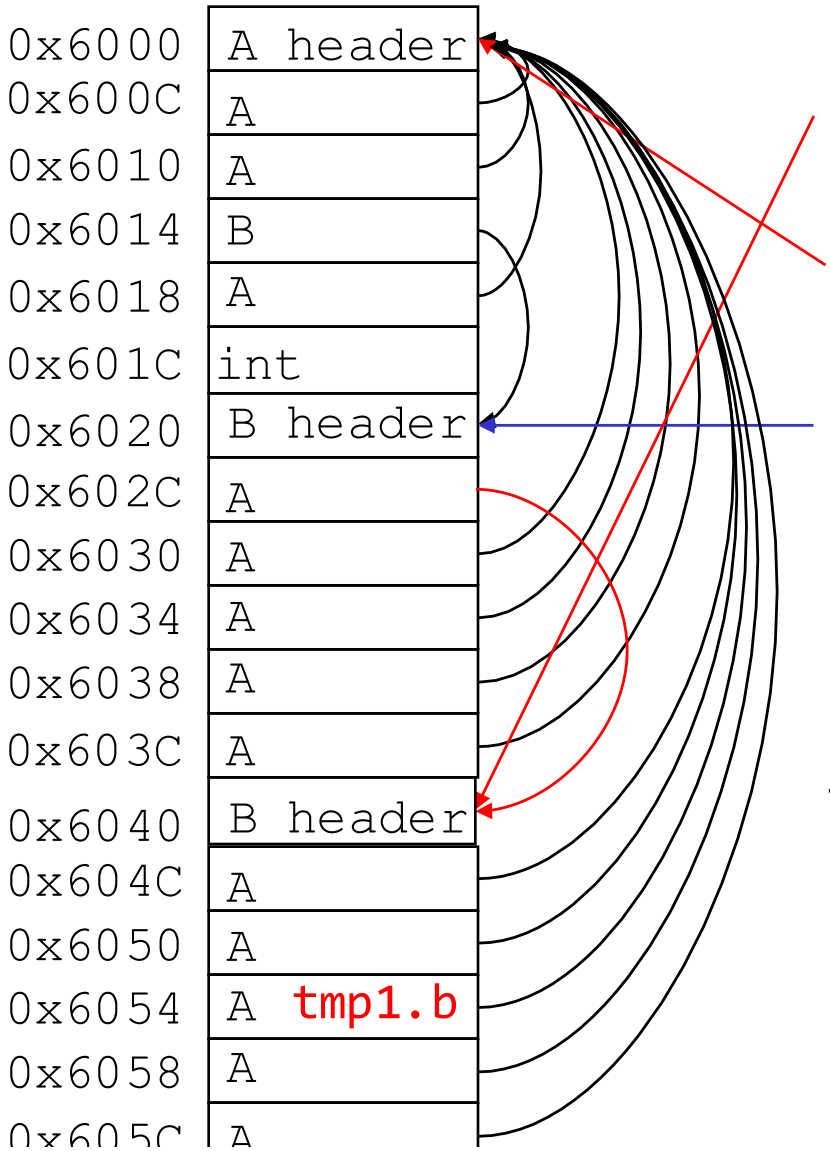
Step 1 (Exploiting The Memory Error)




```
B orig;
A tmp1 = orig.a1;
B bad = tmp1.b;
```

The heap has **one** A object, **many** B objects. All fields of type A point to the only A object that we need here. Place this object close to the many B objects.

Step 1 (Exploiting The Memory Error)



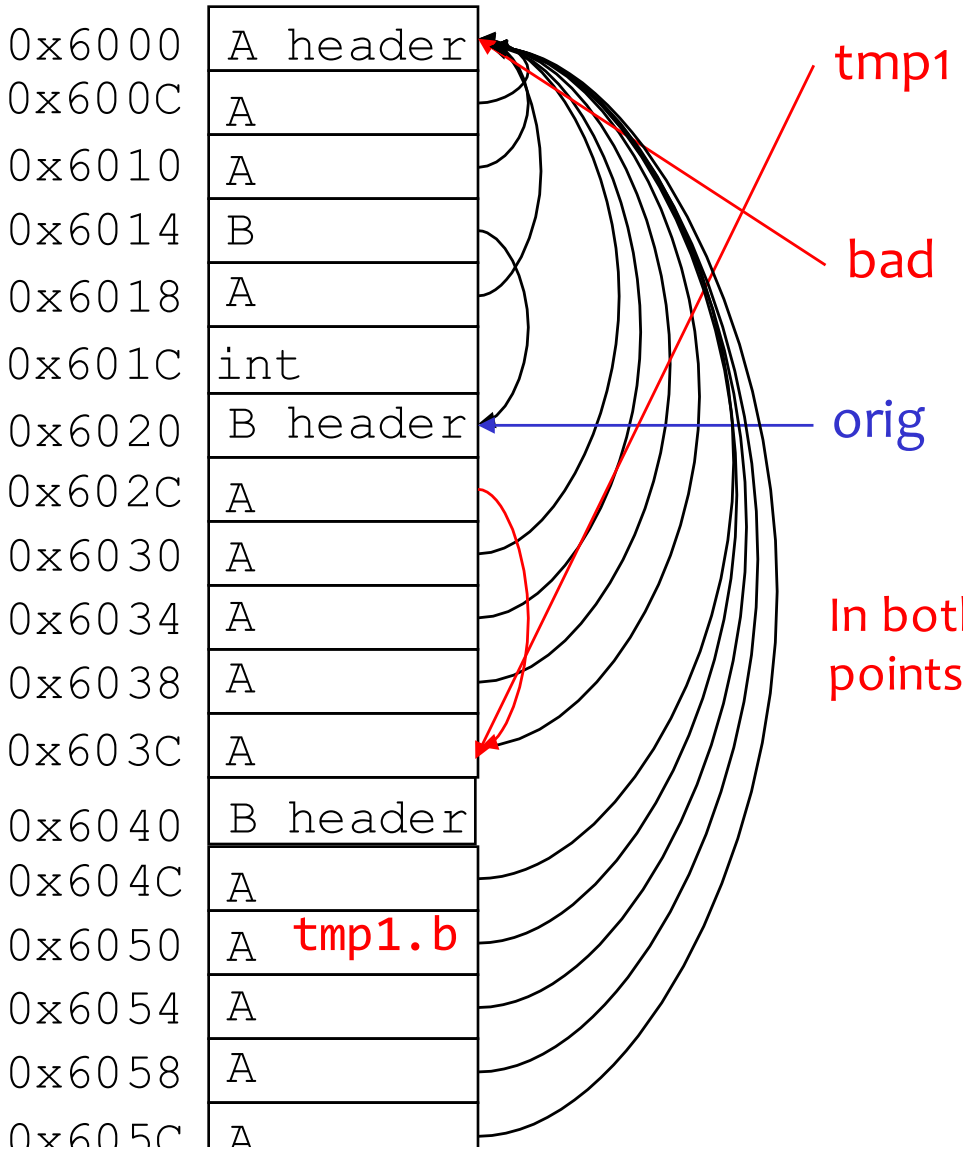
B orig;
A tmp1 = orig.a1;
B bad = tmp1.b;

flip bit 0x40 in orig.a1

Now bad points to an A object!

Note: it is a coincidence that orig.a points to the top of the object header. It could equally likely point into an object of type B.

Step 1 (Exploiting The Memory Error)



B orig;
A tmp1 = orig.a1;
B bad = tmp1.b;

flip bit 0x3C in orig.a1

In both cases, bad, with static type B, now points to an A object!

Step 1 (cont)

Iterate until you find that a flip happened and was exploited.

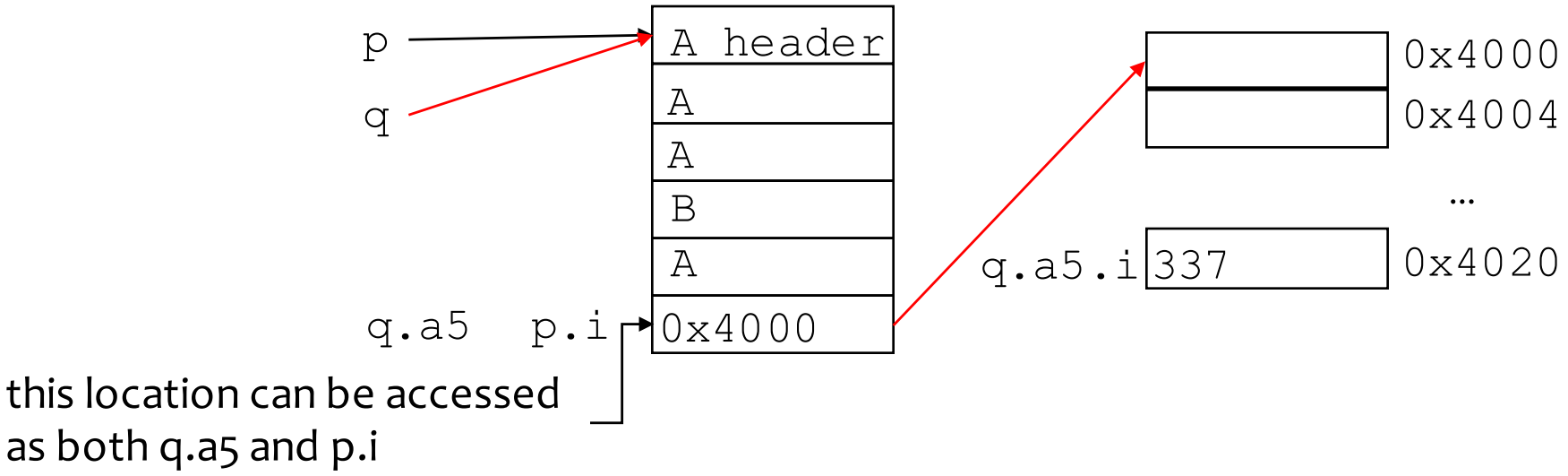
```
A p; // pointer to the single A object
while (true) {
  for (int i = 0; i < b_objs.length; i++) {
    // iterate over all B objects
    B orig = b_objs[i];

    A tmp1 = orig.a1; // Step 1, really check a1, a2, a3, …
    B q = tmp1.b;

    Object o1 = p; Object o2 = q; // check if we found a flip
    // must cast p,q to Object to allow comparison
    if (o1 == o2) {
      writeCode(p,q); // now we're ready to invoke Step 2
} } }
```

Step 2 (Writing arbitrary memory)

```
int offset = 8 * 4;     // Offset of i field in A
A p; B q;               // Initialized in Step 1, p == q;
                        // assume both p and q point to an A

void write(int address, int value) {
  p.i = address - offset;
  q.a5.i = value; // q.a5 is an integer treated as a pointer
}
```

This code type checks

Example: write 337 to address 0x4020



Results (paper by Govindavajhala and Appel)

With software-injected memory errors, took over both IBM and Sun JVMs with 70% success rate

think why not all bit flips lead to a successful exploit

Equally successful through heating DRAM with a lamp

Defense: memory with error-correcting codes

- ECC often not included to cut costs

Most serious domain of attack is smart cards

Paper: http://www.cs.princeton.edu/~sudhakar/papers/memerr.pdf

Summary

- Static types are important
  - Language-based security
  - Performance (requires compiler support)

- But they are only good as long as there are no side channels to circumvent the type system