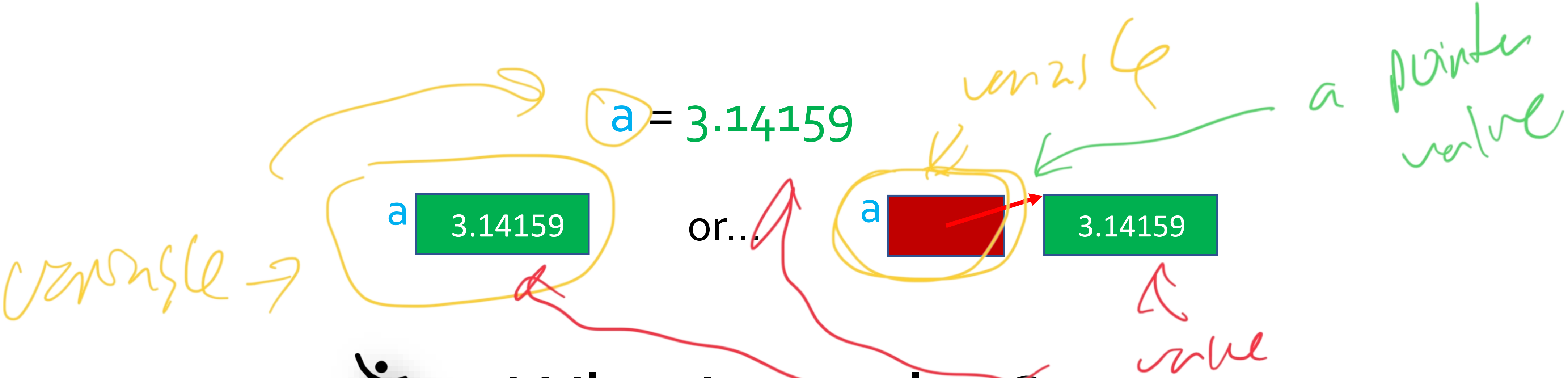
# Data-palooza

In this segment, we're going to focus on how languages manage data (types, variables & values).

Your goal is to be able to pick up a new language and quickly understand how it manages types, variables and values.

# What's a variable?



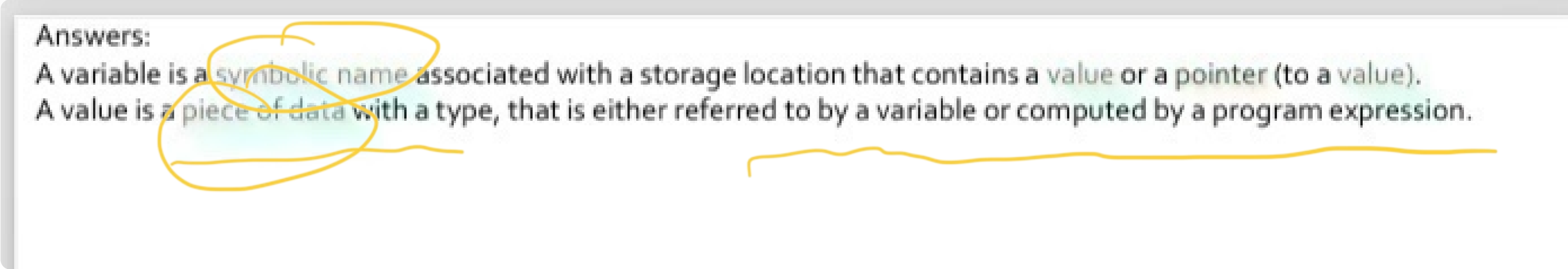
a = 3.14159

a 3.14159

or...

a 3.14159

# What's a value?



Answers:

A variable is a symbolic name associated with a storage location that contains a value or a pointer (to a value).

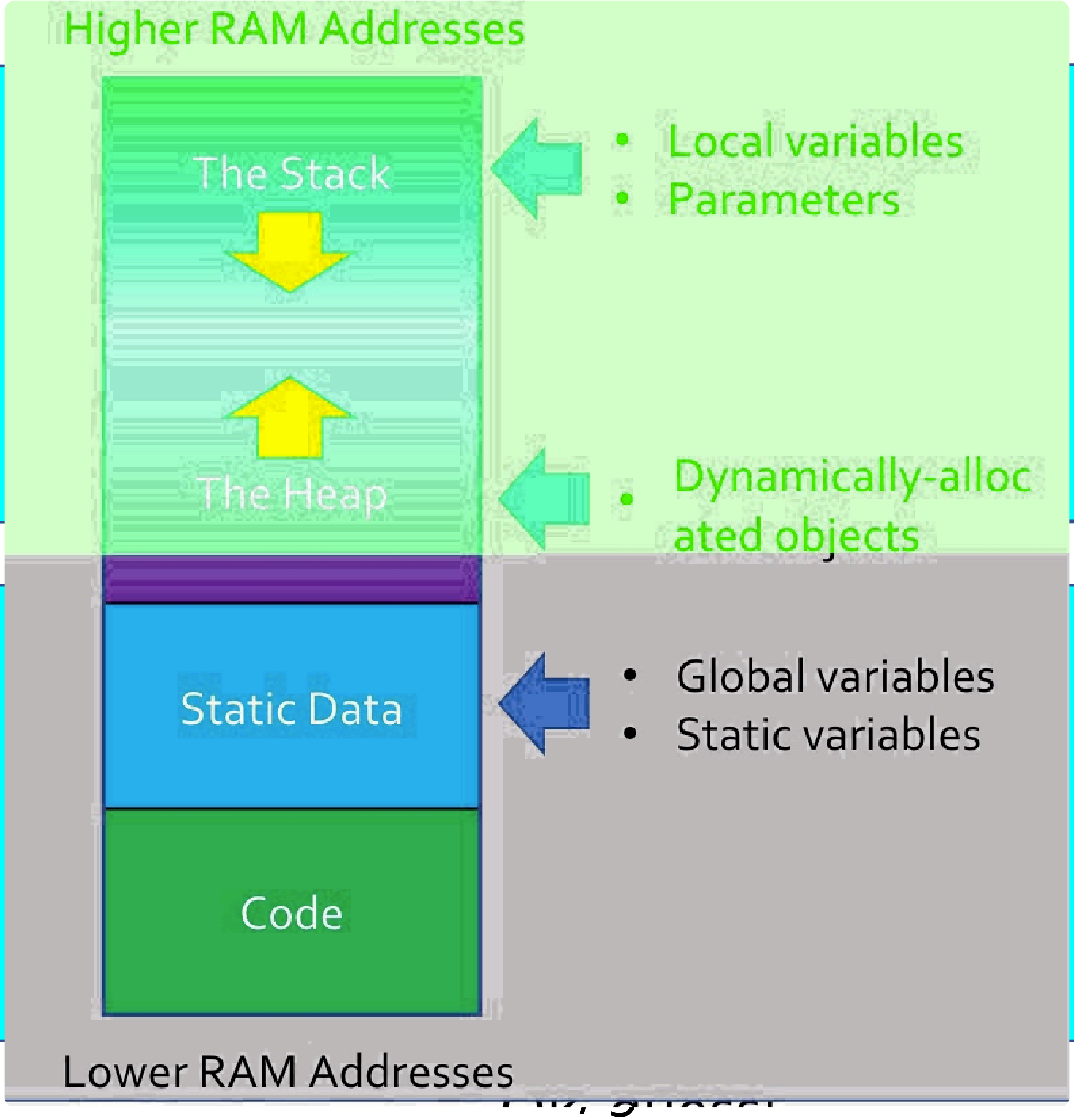
A value is a piece of data with a type, that is either referred to by a variable or computed by a program expression.

# What are all the facets that make up a variable?

I’ll give you some hints…

N???

B??????



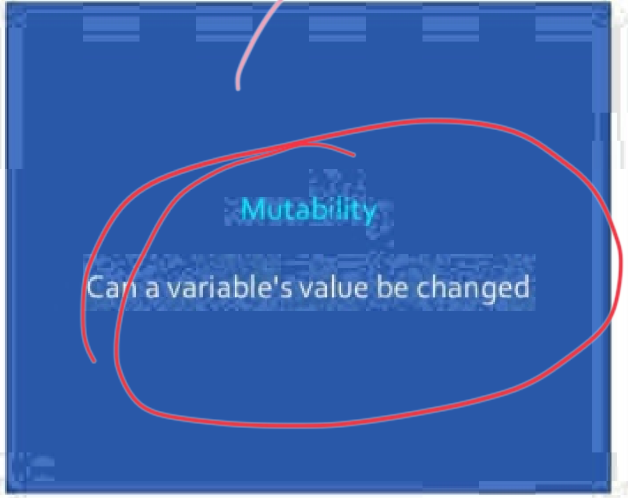
S??????

M?????????

# What are all the facets that make up a variable?

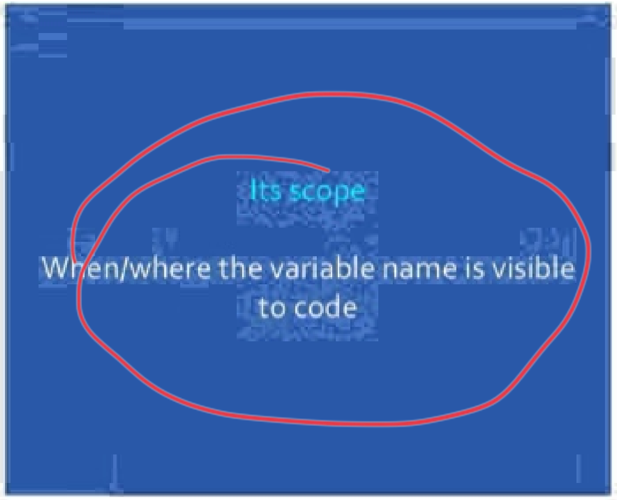
I’ll give you some hints…

const in C++



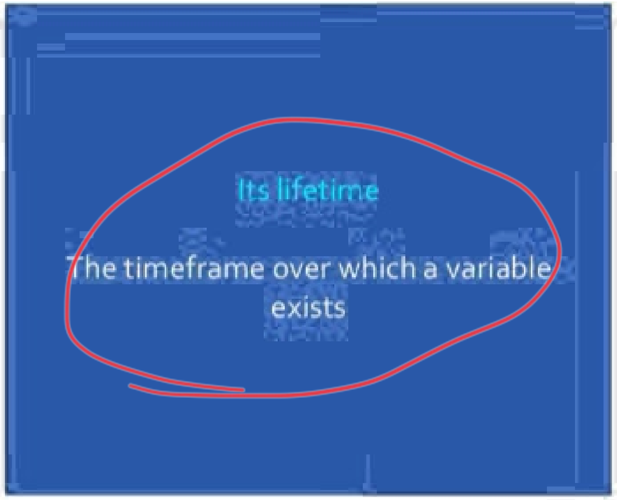
Mutability

Can a variable's value be changed



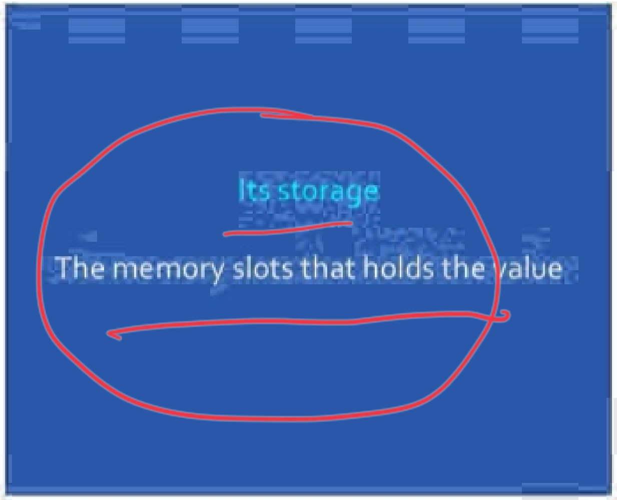
Its scope

When/where the variable name is visible to code



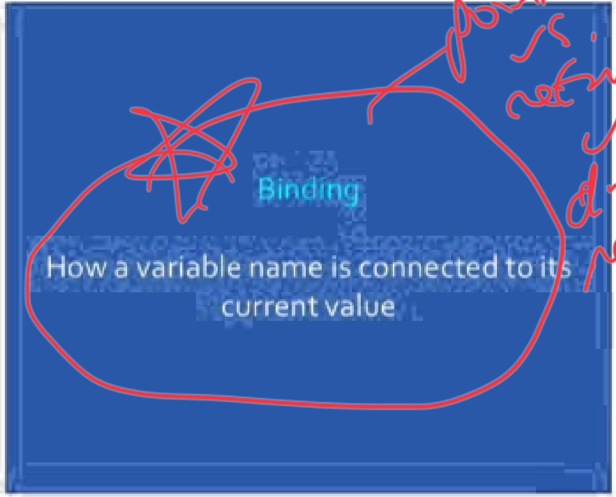
Its lifetime

The timeframe over which a variable exists



Its storage

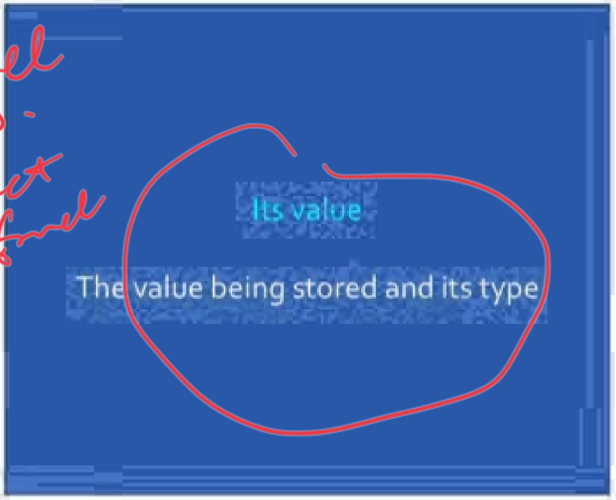
The memory slots that holds the value



Binding

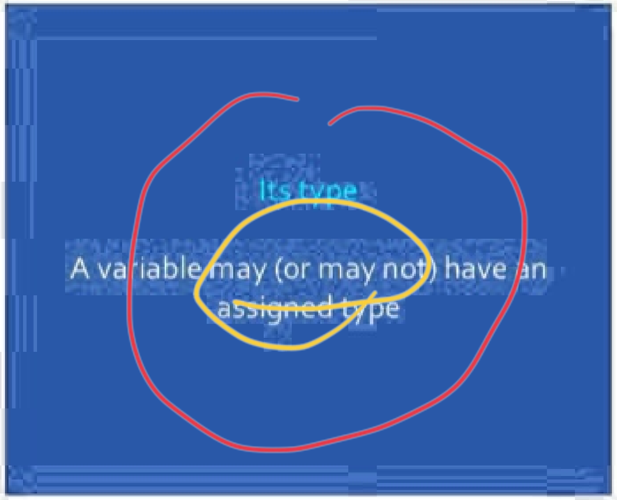
How a variable name is connected to its current value

pointer vs. reference vs. direct reference



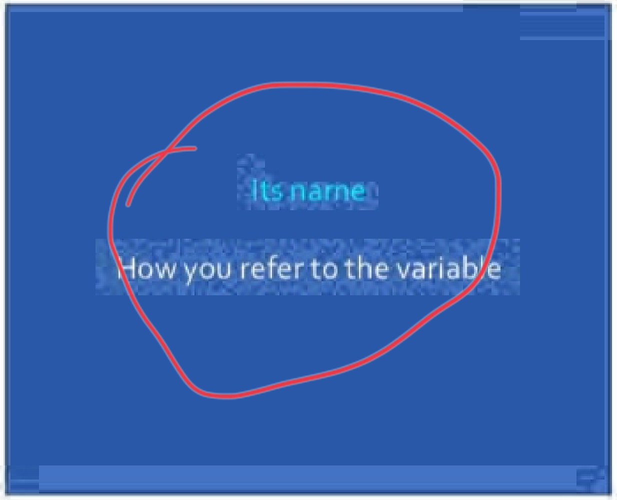
Its value

The value being stored and its type



Its type

A variable may (or may not) have an assigned type



Its name

How you refer to the variable

Ok, guess!

# What are all the facets that make up a value?

Almost all languages stipulate that names should contain valid characters
Almost all languages stipulate that names should not be the same as keywords or constants
Most languages have a rule that disallows spaces in variable names
Some languages have rules about special characters in names, some enforce length restrictions, and some even enforce some sort of case sensitivity rule.

Variable types
What can you infer about a value, given its type?

The set of legal values it can hold
The operations we can perform on it
How much memory you need
How to interpret the bytes stored in RAM
How values are converted between types

# What are all the facets that make up a value?

I'll give you some hints…

~~Its name~~

~~How you refer to the variable~~

Its type

A value always has a type

Its value

The value itself

~~Binding~~

~~How a variable name is connected to its current value~~

Its storage

The memory slots that holds the value

Its lifetime

The timeframe over which a value exists

~~Its scope~~

~~When/where the variable name is visible to code~~

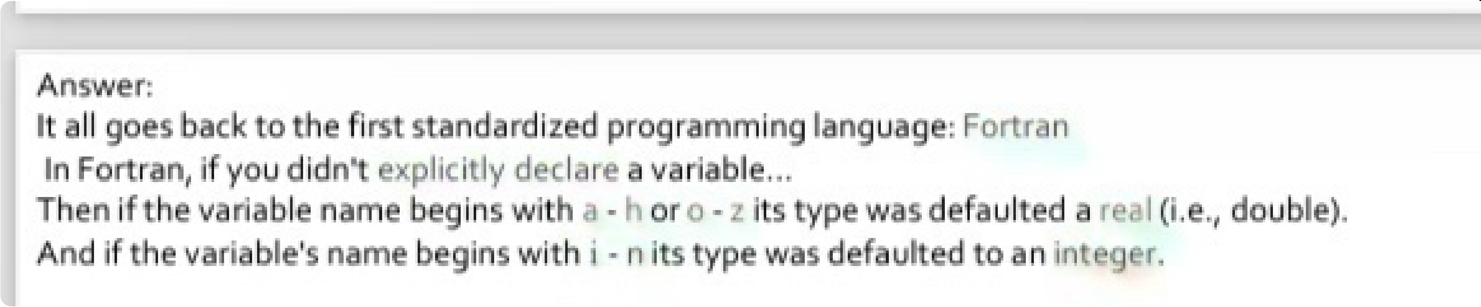
Mutability

Can a value be changed

# Variable Name Trivia!

Question: Why do most loops idiomatically use a variable named i or j for iteration?

```
! compute factorials from 1 to 10
integer nfact
nfact = 1
do i = 1, 10
   nfact = nfact * i
   print*,  i, "! is ", nfact
end do
```



Answer:

It all goes back to the first standardized programming language: Fortran

In Fortran, if you didn't explicitly declare a variable...

Then if the variable name begins with a - h or o - z its type was defaulted a real (i.e., double).

And if the variable's name begins with i - n its type was defaulted to an integer.

Answer:

It all goes back to the first standardized programming language: Fortran

In Fortran, if you didn't explicitly declare a variable...

Then if the variable name begins with a - h or o - z its type was defaulted a real (i.e., double).

And if the variable's name begins with i - n its type was defaulted to an integer.

# Let's do some Deep Dives

## Types

We'll understand how types are used, how languages check for valid types, and how they convert between types

## Scoping and Lifetime

We'll learn how languages decide a variable's scope and lifetime

## Memory Safety

We'll learn how languages safeguard reads/writes to memory

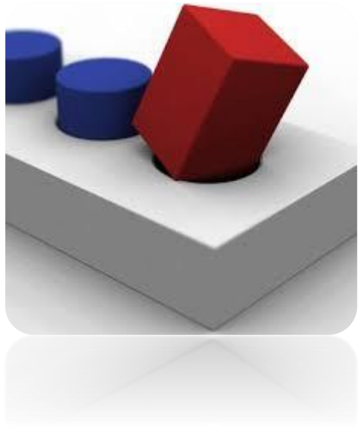
## Mutability

We'll learn how the mutability of variables impacts code correctness

## Binding Semantics

We'll learn how languages associate variable names with values

# Types! Types! Types!



strong vs. weak

dynamic vs. static

By the end of this section, you should be able to:

Take a new language and figure out what kind of typing system it uses.

Understand the implications of that typing system so you can write correct programs in that language.

# What is a Type?

What is a type and what are all the things a type specifies?

Actually, they're not! It is possible to have a language with no types. Assembly languages are one such example of languages with no type system. They just have a register that holds a 32 (or 64) bit value. The value could represent anything (an integer, float, pointer, etc.). BLISS is another example of a language with no types.

# What is a Type?

list()

string can be almost anything, etc.

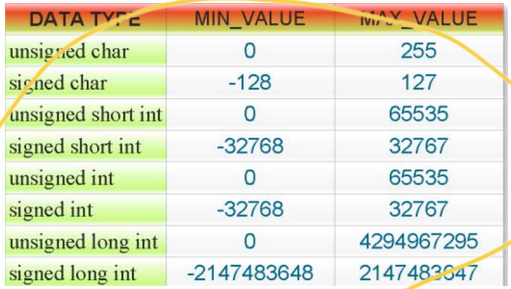
What is a type and what are all the things a type specifies?

enum/ADT might have only a few

A type is a classification that is used to identify a category of data.

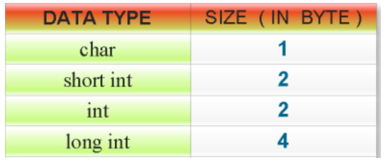
A type defines a range of values, size and encoding, what operations we can perform on it, where it can be used, and how it's converted/cast to other types.

Range of Values

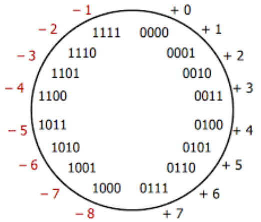


| DATA TYPE | MIN_VALUE | MAX_VALUE |
|---|---|---|
| unsigned char | 0 | 255 |
| signed char | -128 | 127 |
| unsigned short int | 0 | 65535 |
| signed short int | -32768 | 32767 |
| unsigned int | 0 | 65535 |
| signed int | -32768 | 32767 |
| unsigned long int | 0 | 4294967295 |
| signed long int | -2147483648 | 2147483647 |

Size and Encoding



| DATA TYPE | SIZE ( IN BYTE ) |
|---|---|
| char | 1 |
| short int | 2 |
| int | 2 |
| long int | 4 |

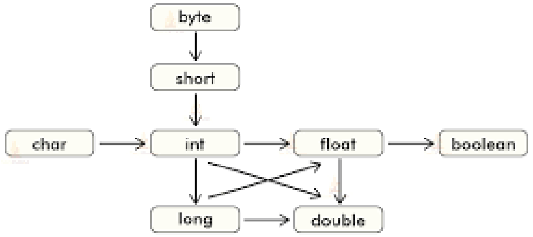


Operations

int: +, -, *, /, ...
bool: &&, ||, !
Nerd: study()

Usage Context

Conversions/Casts



# What Do Languages Use Types For?

when a language infers the type of a variable

### Defining Variables

```
int age = 21;
```

### Type Checking

```
Dog d;
Cat c;
d = c;   // type mismatch
```

### Type Inference

```
var a = 5
print(type(a))  // "int"
```

### Type Conversion

```
float f = 3.14;
int i = f; // conversion
```

### Type Casting

(class + object)

```
Dog d;
Animal *a = &d;   // cast
```

### Polymorphism

```
Dog d;
Animal *a = &d;
a->talk();  // "woof!"
```

### Generics/Templates

```
list<int> stats;
map<string, int> dict;
```

Different

haskell and Python
-> based on context

# Variable Types?

not informed by haskell

In a typed language, must every variable have a type?

Statically typed

```
// C++

void foo() {
  int x;
  ...
}
```

x is bound to a single integer value over its lifetime

```
-- Haskell
f x =
  let exp = 2*3
  in
   x^exp
```

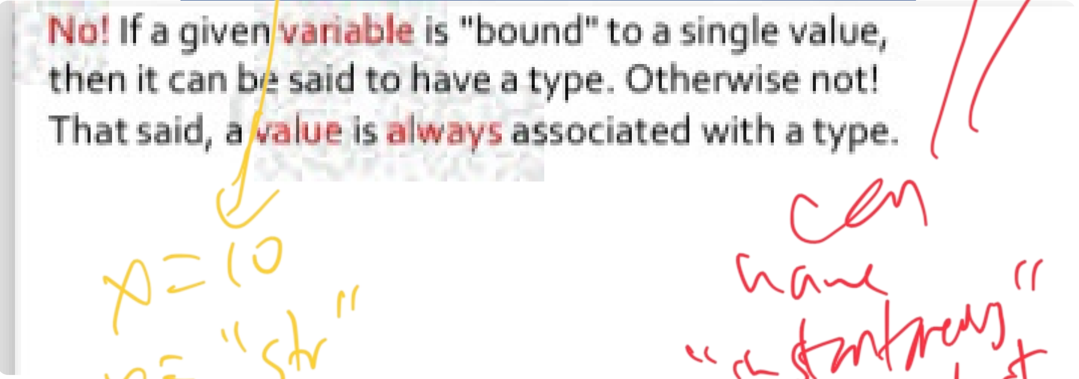
exp is bound to a single integer value as well

statically typed

but can do x=10 x="str" in ghci

```
# Python
def foo(q):
  if q:
     x = "What's my type?"
  else:
     x = 10
```

x is a name w/o type but refers to value of type


No! If a given variable is "bound" to a single value, then it can be said to have a type. Otherwise not! That said, a value is always associated with a type.

can have "arbitrarily" typed but not affixed to

variable types are not fixed → can be bound to many types during lifetime

No! If a given variable is "bound" to a single value, then it can be said to have a type. Otherwise not! That said, a value is always associated with a type.

# Types of Types

Question: How many different types of types can you name?

## Primitives

I???????

F?????

C????

E????

B???????

P???????

## Composites

R??????

U?????

C??????

S??????

T?????

C?????????

## Others

G???????

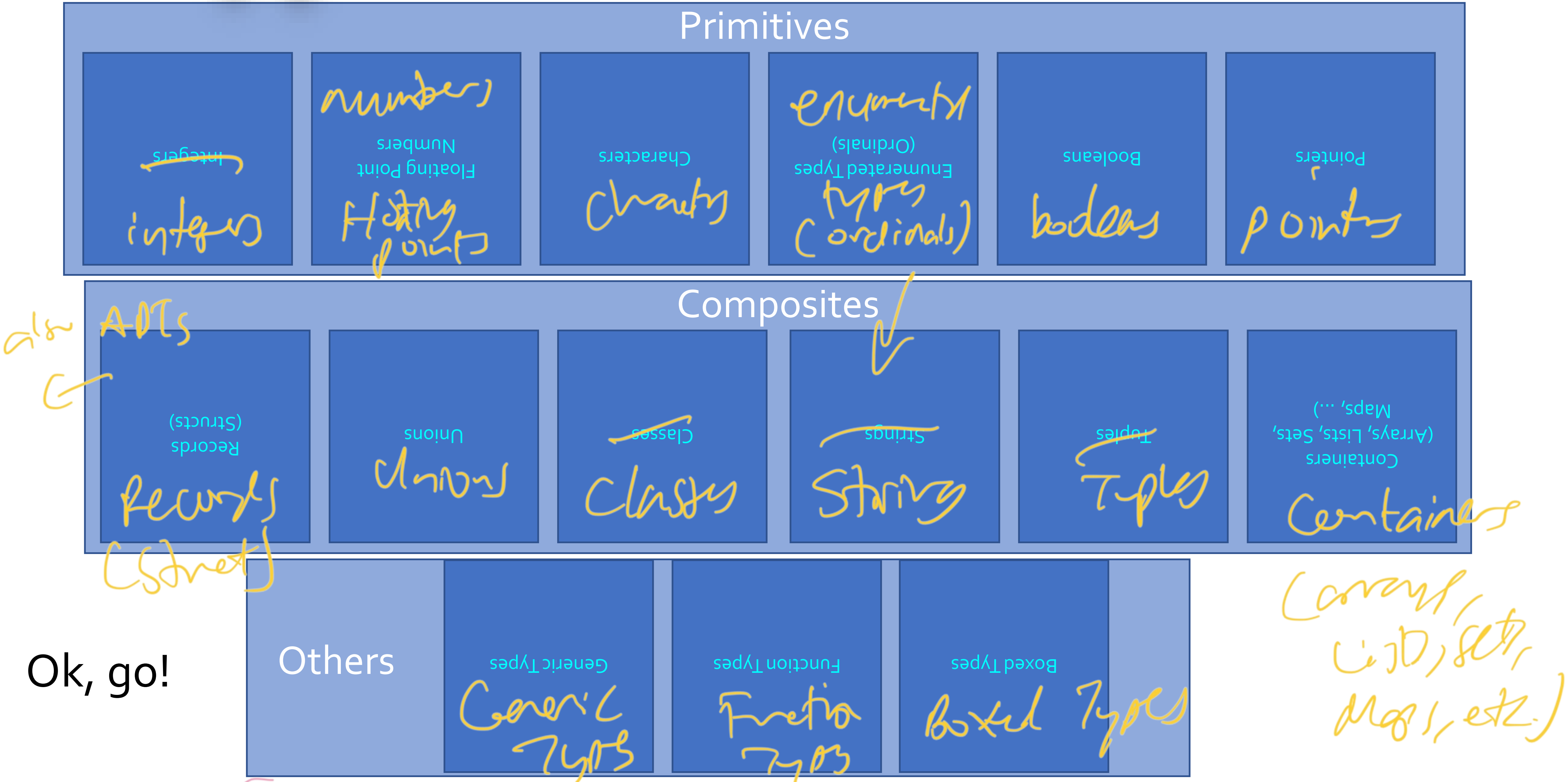
F???????

B????

Ok, go!

# Types of Types

Question: How many different types of types can you name?



## Primitives

~~Integers~~

integers

Numbers
Floating Point

numbers
floating points

Characters

characters

(Ordinals)
Enumerated Types

enumerated types
(ordinals)

Booleans

booleans

Pointers

pointers

## Composites

also ADTs

(Structs)
Records

Records
(struct)

Unions

unions

~~Classes~~

classes

~~Strings~~

Strings

~~Tuples~~

Tuples

Maps, ...)
(Arrays, Lists, Sets,
Containers

Containers

(arrays,
lists, sets,
maps, etc.)

## Others

Generic Types

Generic types

Function Types

Function types

Boxed Types

Boxed types

Ok, go!

# Types of Types

in languages like Python that pass by object reference, this lets you "change primitive type's value!

Question: How many different types of types can you name?

What's a generic type? / Template

A generic type is a type that is parameterized with one or more type parameters, e.g.:

```
template <class T>
class Collection {
public:
  void add(T item) { arr_[count++] = item; }
  ...
private:
  T arr_[MAX_ITEMS];
  int count = 0;
};
```

can mutate member variables but

Haven't heard of boxed types? may be

A boxed type is just an object whose only data member is a primitive (like an int or a double).

```
class Integer {
 public:
   int get() const { return val_; }
 private:
   int val_;
};
```

not a primitive type

wrappers

Haven't heard of unions (aka variants)?

```
union holds_one_of {
 int i; double d; string s;
}

int main() {
  holds_one_of x;
  x.i = 10;     // x holds an int now
  x.s = "Carey" // now x holds a string
}
```

Haven't heard of enumerated types?

```
enum Mood {Happy, Sad, Excited, Silly};

int main() {
 Mood m;
 m = Excited;
 if (m == Sad) cout << "Sorry!";
}
```

(represented as int under the hood

# User-defined Types

classes and types are different

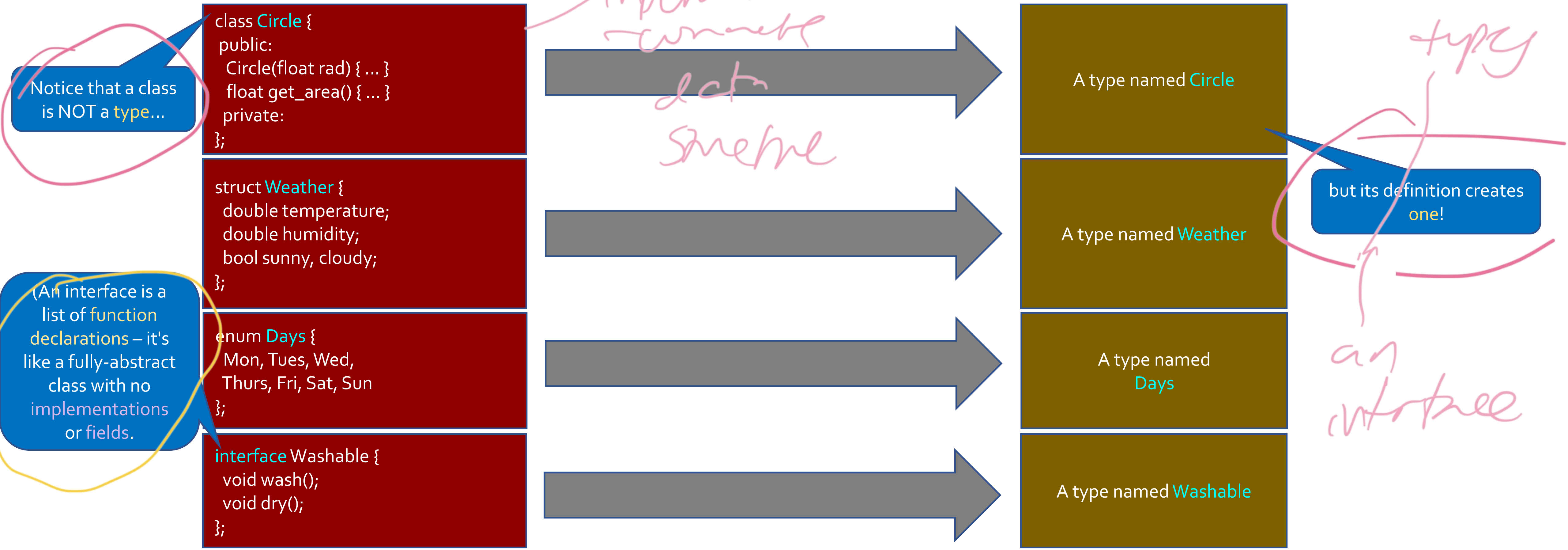
Classes vs. Types

Beyond built-in types like int, double and string...

languages also let users define new types.

For example, every time you define a...

The language implicitly defines...



implementation

concrete

data

scheme

```
class Circle {
 public:
  Circle(float rad) { ... }
  float get_area() { ... }
 private:
};
```

Notice that a class is NOT a type...

A type named Circle

but its definition creates one!

type

```
struct Weather {
  double temperature;
  double humidity;
  bool sunny, cloudy;
};
```

A type named Weather

```
enum Days {
 Mon, Tues, Wed,
 Thurs, Fri, Sat, Sun
};
```

A type named Days

(An interface is a list of function declarations – it's like a fully-abstract class with no implementations or fields.

```
interface Washable {
  void wash();
  void dry();
};
```

A type named Washable

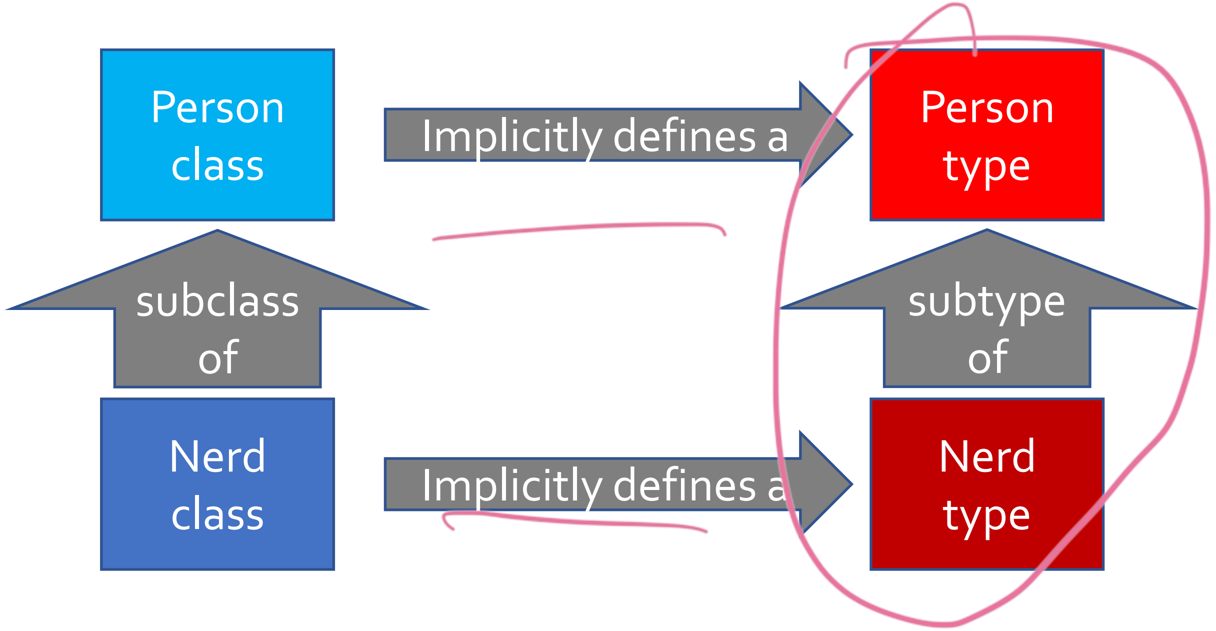
an interface

# Supertypes and Subtypes

As we learned in CS32, some types exhibit a supertype/subtype relationship, where a subtype inherits properties and behaviors from its supertype.

The primary way we define such typing relationships is via class inheritance:

```
class Person {
public:
  virtual void eat()
    { cout << "Nom nom"; }
  virtual void sleep()
    { cout << "Zzzzz"; }
};

class Nerd: public Person {
public:
  virtual void study()
   { cout << "Learn, learn, learn"; }
};
```



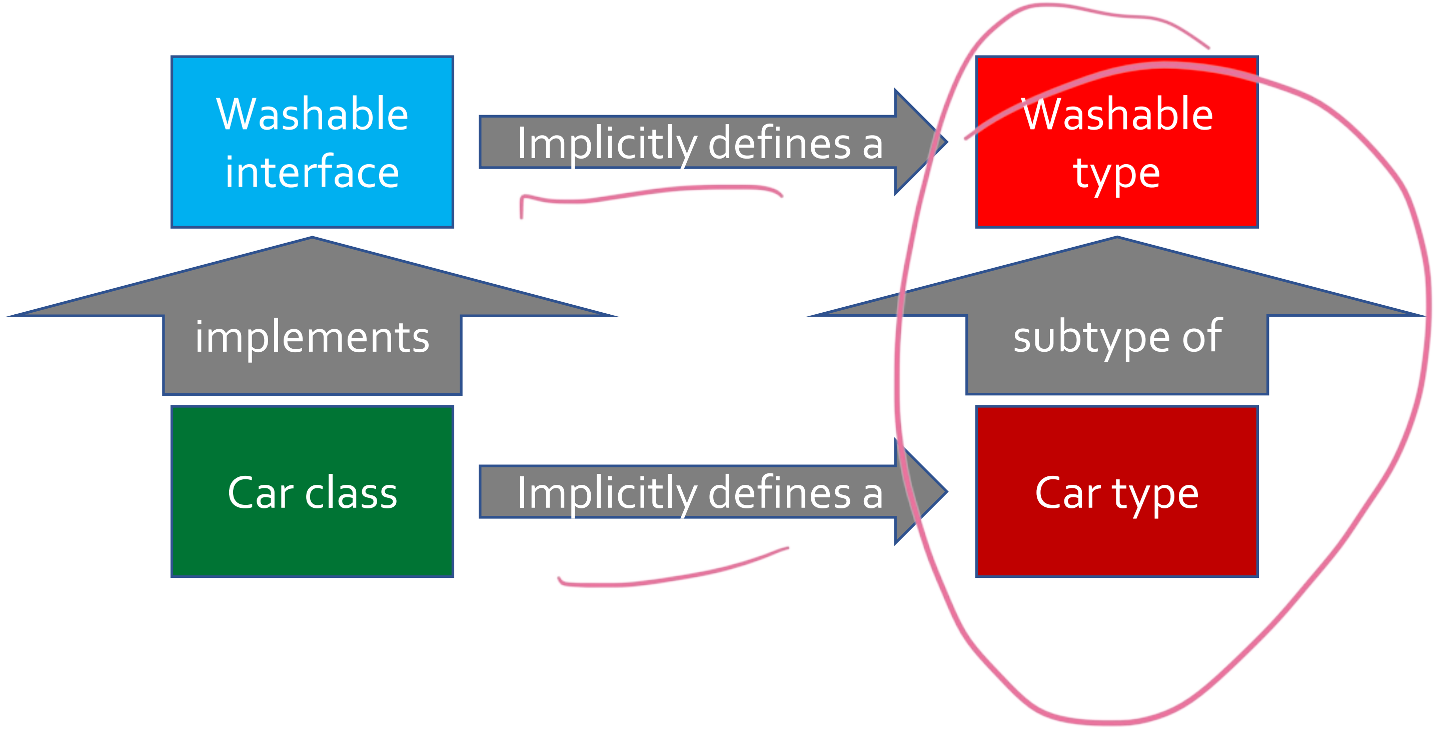
# Supertypes and Subtypes

pure virtual

In addition, we can define supertype/subtype relationships via interface inheritance:

```
class Washable {     // C++ Interface
  virtual void wash() = 0;
  virtual void dry() = 0;
};

class Car: public Washable {
  virtual void wash() {
     cout << "Use soap and water.";
  }
  virtual void dry() {
     cout << "Use dish towel.";
  }
}
```

abstract class



# Supertypes and Subtypes

```
class Person {
public:
  virtual void eat()
    { cout << "Nom nom"; }
  virtual void sleep()
    { cout << "Zzzzz"; }
};

class Nerd: public Person {
public:
  virtual void study()
   { cout << "Learn, learn, learn"; }
};
```

Since we know all Persons can eat and sleep…

we also know all Nerds can eat and sleep!

```
void bePersoney(Person &p) {
  p.eat();
  p.sleep();
}

int main() {
  Nerd nancy;
  bePersoney(nancy);
}
```

This allows us to pass a Nerd…

to a function that accepts Persons…

and know it will support the required operations!

Each subtype has its own unique operations but also inherits all operations from its supertype.

So supertypes/subtypes define not only a type relationship but also an operational relationship as well.

These operational relationships allow languages to support capabilities like subtype polymorphism.

type casting

polymorphism

what operations are possible w/ subtype relation to supertype

# Value Types and Reference Types

Types come in two flavors:

create new

## Value Types

A *value type* is one that can be used to instantiate objects/values (and define pointers/obj refs/references).

```
class Dog {
public:
  Dog(string n) { name_ = n; }
  void bark() { cout << "Woof\n"; }
private:
  string name_;
};
```

An example of a value type would be a type associated with a concrete class (one with all its methods implemented).

Why? Because we can use the type to instantiate objects.

```
Dog d("Kuma"), *p;
```

(and define pointers, etc.)

only use to reference pointer variable but not define whole object

## Reference Types

A *reference type* can only be used to define pointers/object references/references (but *not* instantiate objects/values).

```
class Shape {
public:
  Shape(Color c) { color_ = c; }
  virtual double area() = 0;
private:
  Color color_;
};
```

An example of a reference type would be a type associated with an abstract class (missing some method implementations).

We can only use the type to define pointers/object references!

```
Shape *s;  // Works great!
```

abstract, can't do Shape S (Blame)

# Type Equivalence

Type equivalence is the criteria by which a programming language determines whether two values or variables are of equivalent types.

ex: typescript (variant of JS)

what we're used to

There are two approaches:

## Name Equivalence

creating types called S, T

Two values/variables are of equivalent types only if their type names are identical.

```
// C++: name equivalence
struct S { string a; int b; };
struct T { string a; int b; };

int main() {
   S s1, s2;
   T t1, t2;
   s1 = s2; // this works!
   s1 = t1; // type mismatch error!
}
```

Types S and T are structurally identical!

But they're not considered the same type under name equivalence, so this would be an error.

## Structural Equivalence

Two values/variables are of equivalent types if their structures are identical, regardless of their type names.

```
// typescript: structural equiv.
type S = { a: string; b: number };
type T = { a: string; b: number };

function main() {
   let s1, s2 : S;
   let t1, t2 : T;
   s1 = s2; // this works!
   s1 = t1; // this works too!
}
```

Again, types S and T are structurally identical!

So under structural equivalence, these are considered equivalent types and this would be allowed.

# Type Equivalence

Type equivalence is the criteria by which a programming language determines whether two values or variables are of equivalent types.

There are two approaches:

## Name Equivalence

Two values/variables are of equivalent types only if their type names are identical.

## Structural Equivalence

Two values/variables are of equivalent types if their structures are identical, regardless of their type names.

Most statically typed languages (C++, Java, ...) use name equivalence, while most dynamically typed languages (Python, JavaScript) leverage structural equivalence.

As we go through the various typing systems, look out for the two approaches!

# Type Checking

Let's discuss how languages implement type checking!

And learn the pros and cons of each approach.

# Type Checking Approaches

Compile-time vs. Run-time

C++, Java

Static

Dynamic

Python, JS

Strictness

Strong

Weak

## Static typing

Prior to execution, the type checker determines the type of every expression and ensures all operations are compatible with the types of their operands

## Dynamic typing

As the program executes, the type checker ensures that each primitive operation is invoked with values of the right types, and raises an exception otherwise

# Type Checking Approaches

## Compile-time vs. Run-time

Static Dynamic

Strictness

Strong

### Strong type checking

The language's type system guarantees that all operations are only invoked on objects/values of appropriate types

5 + Dog fails

dog.bark()

Weak

### Weak type checking

The language's type system does NOT guarantee that all operations are invoked on objects/values of appropriate types

cat.bark()

might not be caught

also gradual typing

# Type Checking Approaches

## Compile-time vs. Run-time

compiled

run time

| Strictness | | Static | Dynamic |
|---|---|---|---|
| | Strong | C#, Go,<br>Static typing<br>Haskell, Java, Scala<br>Prior to execution, the type checker determines the type of every expression and ensures all operations are compatible with the types of their operands | Javascript, Perl, PHP, Ruby, Python, Smalltalk |
| | Weak | assembly, C, C++ | NONE that I can find! ☺ |

# What is Static Typing?

With static typing, a type checker checks that all operations are consistent with the types of the operands being operated on prior to the program's execution.

Python

str + str

int + int

~~str + int~~

e.g., the type checker verifies that a and b's types are both compatible with the + operator and with each other.

```
// C++ - explicit types: a, b, and add()
int add(int a, int b) { return a + b; }
```

It can also verify that the type of expression a+b is the same as the return type of the function.

Even though a has no explicit type, Haskell can infer that it must be a numeric type since we're comparing against 0.

Is the same as the type of this returned value!

```
-- Haskell - inferred numeric types
abs a = if a > 0 then a else (-a)
```

The type checker also makes sure this expression (a > 0) is of the Boolean type as required by the if-expression.

The type checker also makes sure the type of this returned value...

If the type checker can't assign distinct types to all variables, functions and expressions and verify type compatibility, then it generates a compiler error.

But if the program type checks, it means the code is (largely) type-safe and few if any checks need to be done at runtime.

unless its weakly typed

# A Precondition for Static Typing?

To support static typing, a language must have a fixed type bound to each variable at its time of definition.

Once a variable's type is assigned, it can't be changed.

Consider C++ (statically typed) and Python (dynamically typed):

The type of variable d is fixed and can't change.

```
// C++
void foo(bool b) {
  double d;
  if (b)
    d = 10.0;
  else
    d = 20.0;

  cout << sqrt(d);
}
```

So the compiler can be sure that sqrt will always be given a value of the right type - before the program even runs!

```
# Python
def foo(b):
  if b:
   d = 10
  else:
   d = "cats"

  print(sqrt(d))
```

Since variable d has no fixed type, it could refer to anything.

So there's no way to verify that sqrt will be passed a value of the right type without running the code!

# Type Inference with Static Typing

Must types be explicitly annotated for static typing?

No! Types can often be inferred! ← haskell

Consider the following program - if we omitted the parameter types, could a compiler infer the types of x and y?

~~int~~ double

string

```
void foo(______ x, ______ y) {
  cout << x + 10;
  cout << y + " is a string!";
}
void bar() {
  double d = 3.14;
  foo(d,"barf");
}
```

if this is only place foo() used

Of course, it's never so simple!

must look at whole program to infer type

So type inference is actually a complex "constraint satisfaction" programming problem!

Languages like Haskell, Go, and now even C++ offer some form of type inference, yet are all statically typed!

# Type Inference: A Few Examples

The auto keyword can be used to infer the variable's type from the right-hand-side expression.

```
// C++ type inference with auto
int main() {
  auto x = 3.14159;
  vector<int> v;
  ...

  for (auto item: v) {
    cout << item << endl;
  }

  auto it = v.begin();
  while(it != v.end()) {
    cout << *it << endl;
    ++it;
  }
}
```

double

item will be inferred to be int.

int

Wow – that simplifies things! It'd otherwise be:

std::vector<int>::iterator it = v.begin();

inferred not still fixed

When using :=, Go infers the type of variables from the right-hand-side expression!

```
// GoLang type inference
func main() {
  msg := "I like languages";
  n := 5
  for i := n; i > 0; i-- {
    fmt.Println(msg);
  }
}
```

```
// Java type inference
public class MyClass {
  public static void main(String args[]) {
      int x=10, y=25;

      var s = "abc";
      var sum = x + y;
  }
}
```

If you use the var keyword, Java also infers the type of variables!

# In Static  Typing, Is There Ever a Need to Check Types at Runtime?

Yes!  Even in statically-typed languages,
sometype checking must be done at runtime!

For example, when we down-cast!

This is a downcast    it says: "I want to treat our p variable as if it refers to a Student object."

error: invalid downcast from Doctor to Student

```
class Person { ... };
class Student : public Person { ... };
class Doctor : public Person { ... };

void partay(Person &p) {
  // assumes only students go to parties
  Student &s = dynamic_cast<Student &>(p);
  s.getDrunkAtParty();
}

int main() {
  Doctor d("Dr. Fauci");
  partay(d);
}
```

At the instant this downcast happens, C++ knows it's operating on a Person... but it doesn't know what type of person.

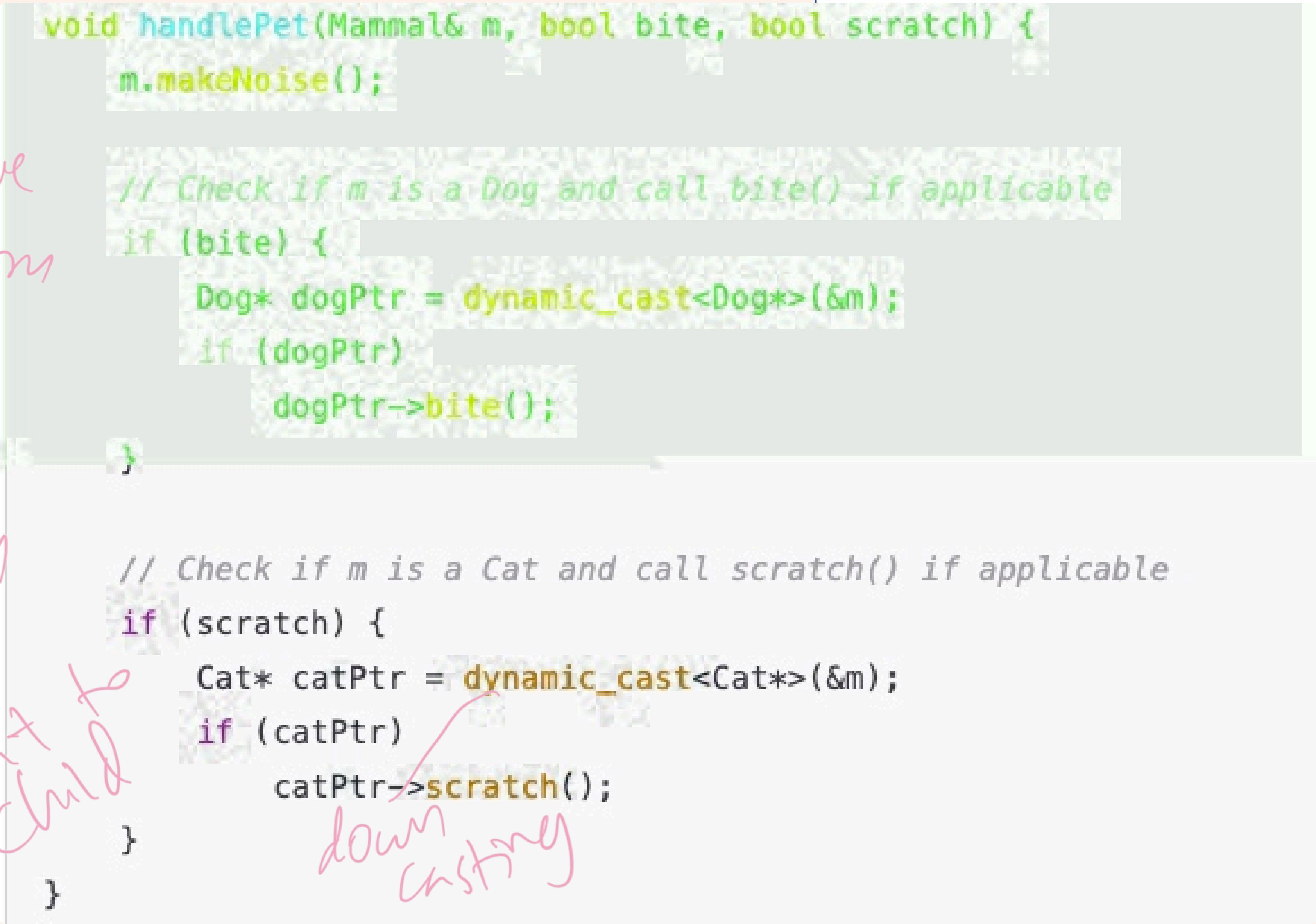
So C++ checks in real-time whether the object passed in is compatible with the downcast (is this Person really a Student?).

If not, the runtime type checker throws an exception.

C++ weakly typed so might allow a bit of static check

# Static Type Checking is Conservative

```
class Mammal {
public:
  string name()
  virtual void
};
class Dog: pub
public:
  void   makeN
  void bite() {
};
class Cat: publ
public:
  void   makeN
  void scratch()
};

void   handleP
  m.makeNoise
  if (   m.name(
     m.bite  ();
  else (   m.nan
     m.scratch
}
```



```
void handlePet(Mammal& m, bool bite, bool scratch) {
    m.makeNoise();

    // Check if m is a Dog and call bite() if applicable
    if (bite) {
      Dog* dogPtr = dynamic_cast<Dog*>(&m);
      if (dogPtr)
        dogPtr->bite();
    }

    // Check if m is a Cat and call scratch() if applicable
    if (scratch) {
      Cat* catPtr = dynamic_cast<Cat*>(&m);
      if (catPtr)
        catPtr->scratch();
    }
}
```

compiler can catch

event technically
compiling!

e type safety the
ly conservative.

hich only asks
to scratch...

ecking because
scratch() methods!

typename

typecasting

can only cast from parent to child

down casting

# Static Type Checking Pros and Cons

## What are the pros of static type checking?

Produces faster code (fewer type checks at runtime, optimizations possible)

Produces faster code (since we don't have to check during run time)

Detects bugs earlier in development

allows for earlier bug detection (at compile time)

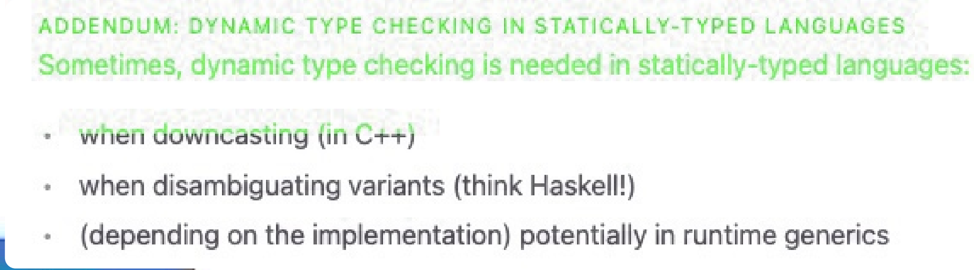
No need to write custom code to check types

no need to write custom type checks

type

fewer type checks, optimizations

## What are the cons of static



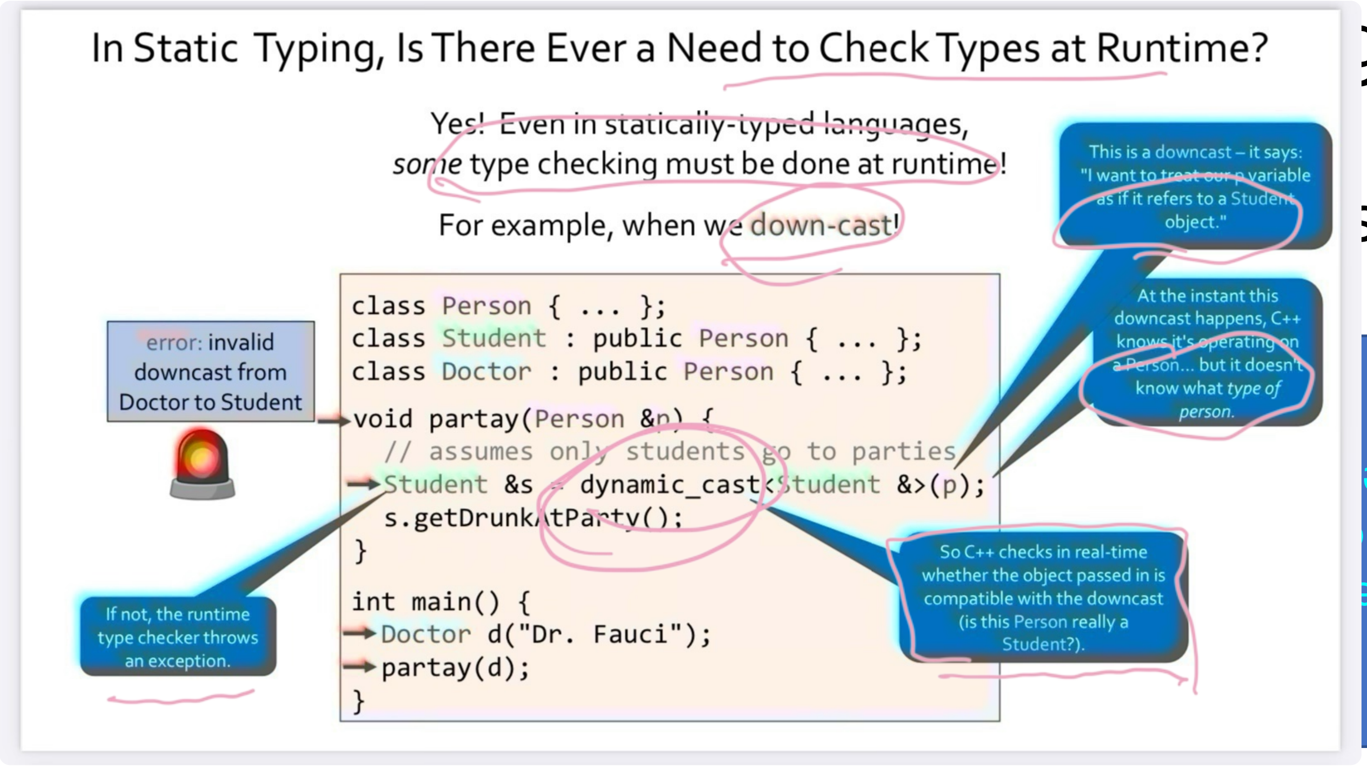
ADDENDUM: DYNAMIC TYPE CHECKING IN STATICALLY-TYPED LANGUAGES

Sometimes, dynamic type checking is needed in statically-typed languages:

- when downcasting (in C++)
- when disambiguating variants (think Haskell!)
- (depending on the implementation) potentially in runtime generics

Static type checking is conservative and may error-out on perfectly valid code

Static typing requires a type checking phase before execution, which can slow development



## In Static Typing, Is There Ever a Need to Check Types at Runtime?

Yes! Even in statically-typed languages, *some* type checking must be done at runtime!

For example, when we down-cast!

```
class Person { ... };
class Student : public Person { ... };
class Doctor : public Person { ... };

void partay(Person &p) {
  // assumes only students go to parties
  Student &s = dynamic_cast<Student &>(p);
  s.getDrunkAtParty();
}

int main() {
  Doctor d("Dr. Fauci");
  partay(d);
}
```

This is a downcast – it says: "I want to treat our p variable as if it refers to a Student object."

At the instant this downcast happens, C++ knows it's operating on a Person... but it doesn't know what *type of person.*

So C++ checks in real-time whether the object passed in is compatible with the downcast (is this Person really a Student?).

If not, the runtime type checker throws an exception.

error: invalid downcast from Doctor to Student

# g Pros and Cons

## static type checking?

De

de

No need to write custom code to check types

## What are the cons of static type checking?

Static type checking is conservative and may error-out on perfectly valid code

Static typing requires a type checking phase before execution, which can slow development

compiler can prevent and make it seem strongly typed

undefined

behavior instead of exception

# Type Checking Approaches

## Compile-time vs. Run-time

| | | Static | Dynamic |
|---|---|---|---|
| Strictness | Strong | C#, Go, Haskell, Java, Scala | Dynamic typing<br>As the program executes, the type checker ensures that each primitive operation is invoked with values of the right types, and raises an exception otherwise |
| | Weak | Assembly Language, C, C++ | |

# Dynamic Typing

In a dynamically-typed language, the safety of operations on variables/values is checked as the program runs rather than at compile time. [illegible]

If the code attempts an illegal operation on a value, an exception is generated or the program crashes.

```
def add(x,y):
 print(x + y)

def foo():
 a = 10
 b = "cooties"
 add(a,b)
```

```
TypeError: unsupported operand
type(s) for +: 'int' and 'str'
```

```
AttributeError: 'Lion' object has no
attribute 'quack'
```

```
def do_something(x):
 x.quack()

def main():
 a = Lion("Leo")
 do_something(a)
```

# Dynamic Typing: Origin Story

Dynamic type checking was pioneered in the LISP language back in 1958.

don't really have type

For flexibility, John McCarthy designed LISP so that variables weren't required to have a fixed type, e.g.

Why? Their types depend upon run-time conditions which aren't predictable at compile time!

named

```
(setq x 1)
(if (== some_condition True)
    (setq y 6)
    (setq y "hi"))
```

refer to values w/ types

But he had a problem - the static type checking approach only works when variables have fixed types.

```
(add x y)
```

There's no way a compiler can determine if both operands are compatible!

types

So he needed a different kind of type checking.

# Dynamic Typing: Types Associated with Values!

As with LISP, in today's dynamically-typed languages, we typically don't assign fixed types to variables.

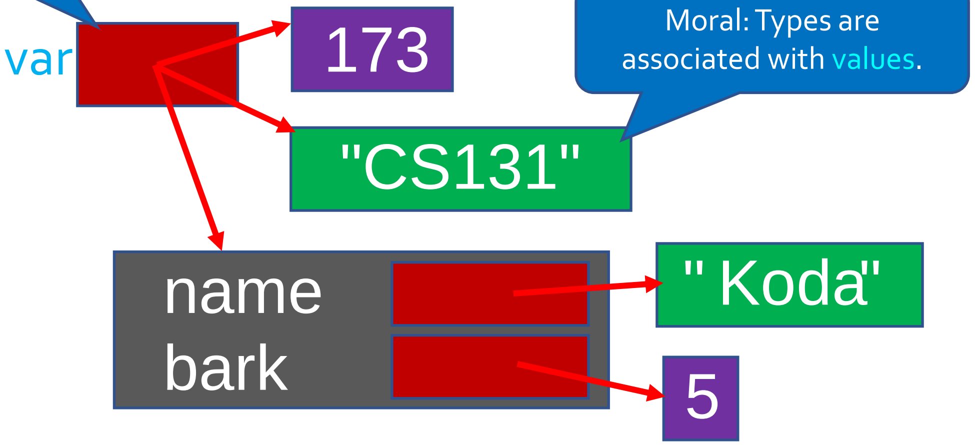
Because of this, we say that in dynamically typed languages:
"types are associated with values and not variables"

For example, Python variables don't have fixed types!

```
def main():
  var = 173
  var = "CS131"
  var = Dog("Koda", 5)
```

A variable can refer to values of different types over time!

NOT variables!



Moral: Types are associated with values.

# How is Dynamic Type Checking Performed?

If *variables* don't have types, how can a dynamically-typed language perform type checking at runtime?

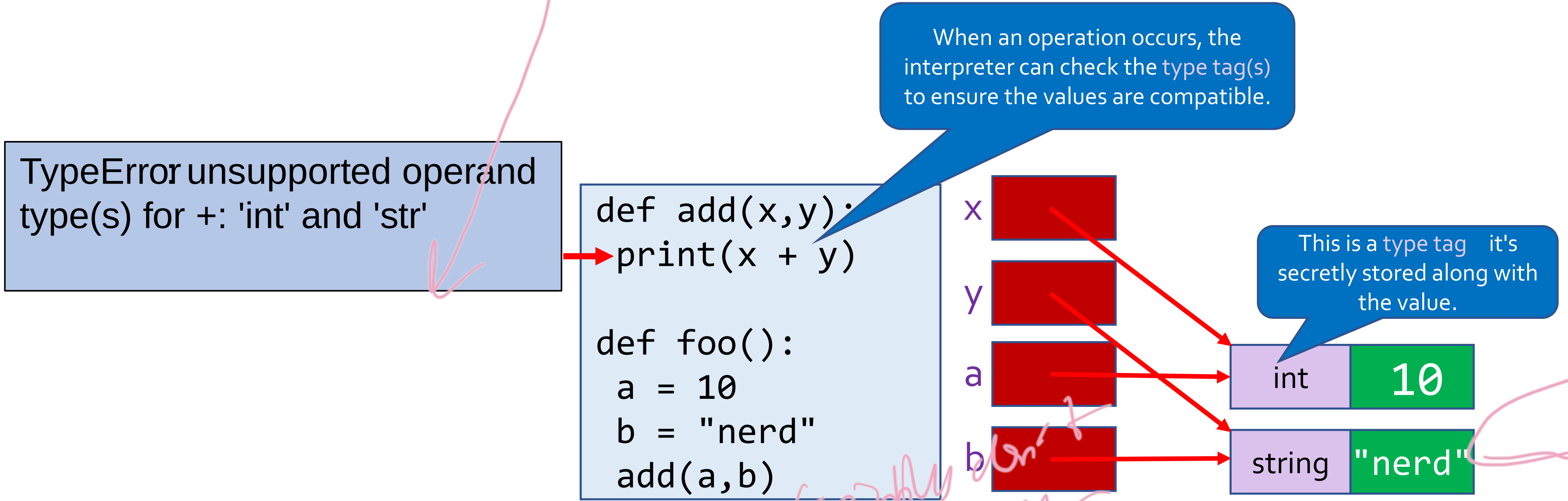
# How is Dynamic Type Checking Performed?

If variables don't have types, how can a dynamically-typed language perform type checking at runtime?

Python has this but not enforced

Answer: The compiler/interpreter stores type information (called a type tag) along with every value/object!

This type information is used to check all operations!



When an operation occurs, the interpreter can check the type tag(s) to ensure the values are compatible.

TypeError: unsupported operand type(s) for +: 'int' and 'str'

```
def add(x,y):
 print(x + y)

def foo():
 a = 10
 b = "nerd"
 add(a,b)
```

This is a type tag it's secretly stored along with the value.

Variables don't have types

# Dynamic Typing: A Few Examples

Here's a function that prints out value v a total of n times, with strings in quotes:

#3: referred to by a variable!

#1: This is called type introspection. It can be used by a function to...

```
-- Lua language
function print_n (v, n)
  for i = 1, n do
    if type(v) == "string" then
      print('"' .. v .. '"')
    else
      print(value)
    end
  end
end

print_n("Hello", 3)
print_n(42, 2)
```

```
# Ruby Language
def print_n (value, n)
  n.times do
    if value.is_a?(String)
      puts "\"#{value}\""
    else
      puts value
    end
  end
end

print_n("Hello", 3)
print_n(42, 2)
```

```
# Julia language
function print_n (v, n:: Int )
  for i in 1:n
    if isa (v, String )
      println ("\"$v\"" )
    else
      println (v)
    end
  end
end

print_n ("Hello" , 3)
print_n (42, 2)
```

#2: determine the type of a value...

#4: This is called a type annotation.
It tells the program that only ints can be passed to the second parameter. But nothing prevents you from changing n's value later, e.g.: n = "ha!"

# Let's Quack!

Consider the following three classes and the code below which uses them.

```
class  PersonInDuckSuit:
  ...
  def  quack(self):
    print('Hi! Uh... I mean quack.')

class  Duck:
  ...
  def  quack(self):
    print('Quack quack quack!')

class  Vehicle:
  ...
  def  drive(self):
    print('Vroooom!')
```

```
def  quack_please(x):
  x.quack()

p = PersonInDuckSuit()
d = Duck()
v = Vehicle()
quack_please(p)
quack_please(d)
quack_please(v)
```

What does this program print?

# Let's Quack!

Consider the following three classes and the code below which uses them.

```
class PersonInDuckSuit:
  ...
  def quack(self):
    print('Hi! Uh... I mean quack.')

class Duck:
  ...
  def quack(self):
    print('Quack quack quack!')

class Vehicle:
  ...
  def drive(self):
    print('Vrooooom!')
```

```
def quack_please(x):
  x.quack()

p = PersonInDuckSuit()
d = Duck()
v = Vehicle()
quack_please(p)
quack_please(d)
quack_please(v)
```

#1: Since variable x could refer to virtually any type of object/value...

#2: at the moment of execution, the type checker verifies that the target object has a quack() method...

#3: and if it does, it calls it!

#4: and if not, it generates a runtime error.

statically typed and need a type for x

instead of bug

Duck

or inherited

R, it's a subclass

an duck

behavior of methods

key

What does this program print?

```
Hi! Uh... I mean quack.
Quack quack quack!
AttributeError: 'Vehicle'
object has no attribute 'quack'
```

Neat! As long as an object has a quack method, the quack_please function just works with it!

And notice, our classes are totally unrelated (i.e., no inheritance)!

# Duck Typing in Other Languages

Ruby, which is dynamically typed, also offers duck typing. Let's see!

```
# ruby duck typing

class Duck
 def quack
  puts "Quack, quack"
 end
end

class Dog
 def quack
  puts "Woof... I mean quack!"
 end
end

animals = [Duck.new,Dog.new]
animals.each do |animal|
 animal.quack()
end
```

And here's an example from JavaScript!

```
// JavaScript duck typing
var cyrile_the_duck = {
  swim: function ()
          { console.log("Paddle paddle!"); },
  color: "brown"
};

var michael_phelps = {
  swim: function ()
          { console.log("Back stroke!"); },
  outfit: "Speedos"
};

function process(who) {
  who.swim();
}

process(cyrile_the_duck);  // Paddle paddle!
process(michael_phelps);   // Back stroke!
```

Academic Robot Says:

"I'd argue that Duck Typing is a form of structural typing!

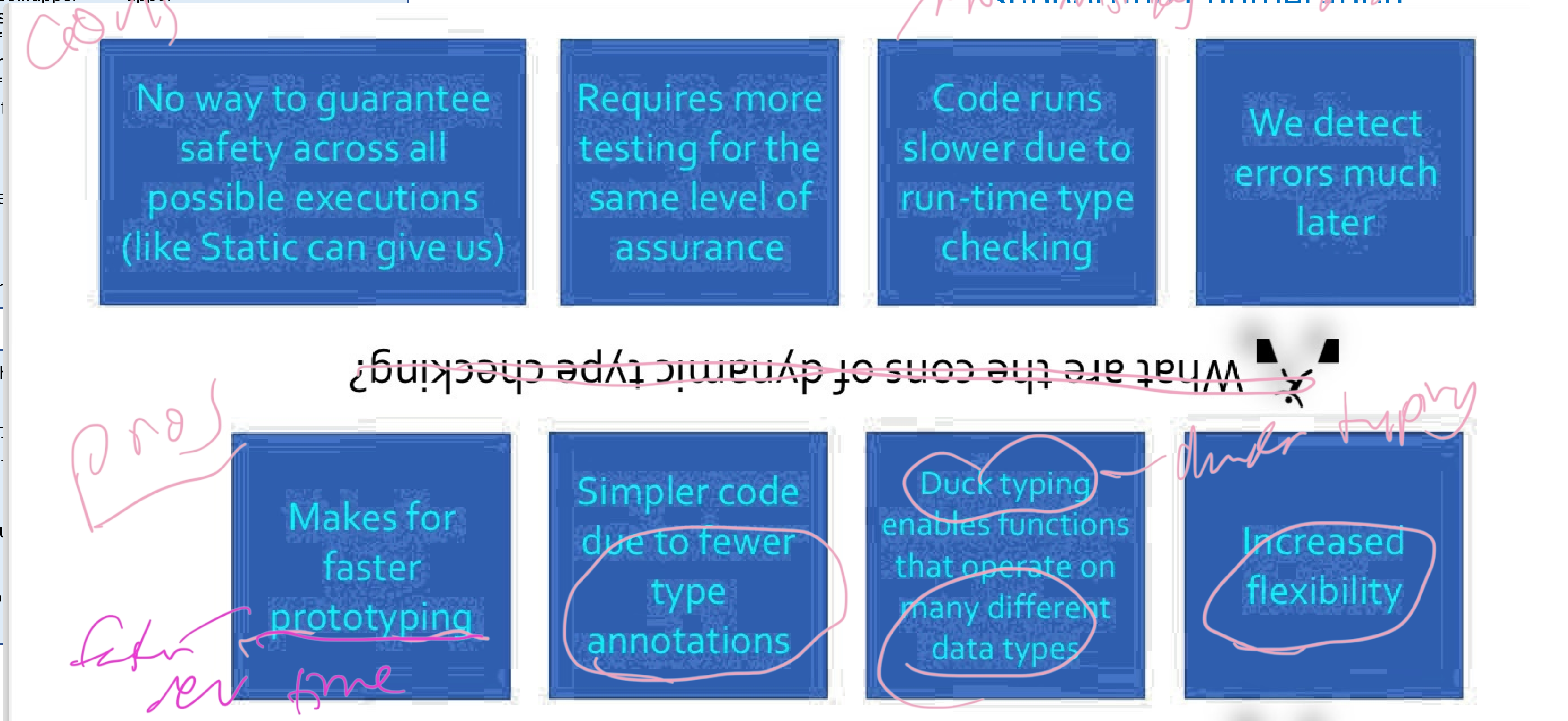
Prove me wrong!"

# Duck Typing: Cool Uses from Python

```
# Python duck typing for iteration
class Cubes:
    def __ init __(self,     lower ,   upper ):
        self.upper      = upper
        s
    def
        r
    def
        i



        e


for   i
    prir
```

```
# Pytl
class
  def _
    sel
    sel

  def
    retu


d = D
print
```

Supporting Enumeration



cons

more mistakes

more space required

No way to guarantee safety across all possible executions (like Static can give us)

Requires more testing for the same level of assurance

Code runs slower due to run-time type checking

We detect errors much later

~~What are the cons of dynamic type checking?~~

pros

Makes for faster ~~prototyping~~

faster dev time

Simpler code due to fewer type annotations

Duck typing enables functions that operate on many different data types

duck typing

Increased flexibility

# Dynamic Type Checking Pros and Cons

What are the pros of dynamic type checking?

What are the cons of dynamic type checking?

# Dynamic Type Checking Pros and Cons

## What are the pros of dynamic type checking?

- Increased flexibility
- Duck typing enables functions that operate on many different data types
- Simpler code due to fewer type annotations
- Makes for faster prototyping

## What are the cons of dynamic type checking?

- We detect errors much later
- Code runs slower due to run-time type checking
- Requires more testing for the same level of assurance
- No way to guarantee safety across all possible executions (like Static can give us)

# A Hybrid Type Checking Approach: Gradual Typing

## Static typing

Prior to execution, the type checker determines the type of every expression and ensures all operations are compatible with the types of their operands

## Gradual typing

Some variables may be given explicit types, others may be left untyped.

Type checking occurs partly before execution and partly during runtime.

## Dynamic typing

As the program executes, the type checker ensures that each primitive operation is invoked with values of the right types, and raises an exception otherwise

We've just learned the differences between static and dynamic typing.

There's actually a less well-known hybrid approach also worth briefly discussing: gradual typing

Languages like PHP and TypeScript use it – so it's worth a quick discussion!

# Gradual Typing

x has *no* type

```
def square(x):
    return x * x

result = square("foo")
```

x has a type

```
def square(x : int):
    return x * x

result = square("foo")
```

With gradual typing, you can choose whether to specify a type for variables/parameters.

If a variable is untyped, then type errors for that variable are detected at runtime!

But if you do specify a type, then *some* type errors can be detected at compile time!

OK, but what happens if we pass an untyped variable to a typed variable?

We pass an untyped variable y

```
def square(x : int):
    return x * x

def what_happens(y):
  print(square(y))
```

to a typed parameter

Challenge: Will a gradually typed language allow this? Why or why not?

# Gradual Typing

x has *no* type

```
def square(x):
    return x * x

result = square("foo")
```

x has a type

```
def square(x : int):
    return x * x

result = square("foo")
```

With gradual typing, you can choose whether to specify a type for variables/parameters.

If a variable is untyped, then type errors for that variable are detected at runtime!

But if you do specify a type, then *some* type errors can be detected at compile time!

OK, but what happens if we pass an untyped variable to a typed variable?

We pass an untyped variable y

```
def square(x : int):
    return x * x

def what_happens(y):
  print(square(y))
```

to a typed parameter

Challenge: Will a gradually typed language allow this? Why or why not?

Answer: You may pass an untyped variable or expression to a typed variable and it'll compile fine!

Since you could pass an invalid type, the program will check for errors at runtime!

# Classify That Language: Type Checking

Ok, let's test our understanding of static, dynamic and gradual typing!

```
fun greet(name: String) {
  print("Hello, $name!")
}


fun main() {
  var n = "Graciela";
  greet(n);

  n = 10;
}
```

Compiler: The integer literal does not conform to the expected type String

type inference

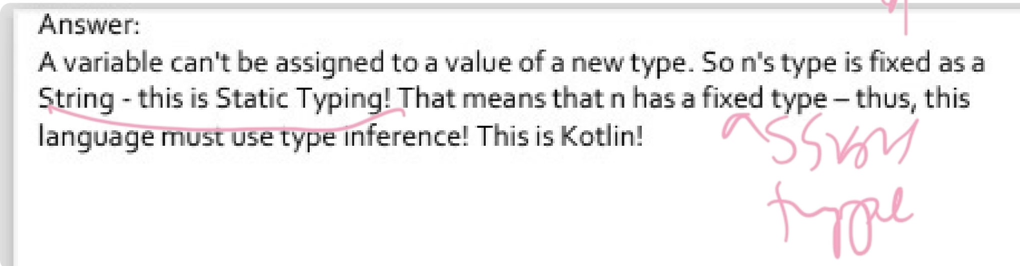
static typing, cannot convert string to int

The following program generates a single compilation error.

Is this language statically, dynamically, or gradually typed?

ex: maybe n = "Graciela" makes it no type, while var n infers and assigns type

(could be gradual → can't tell

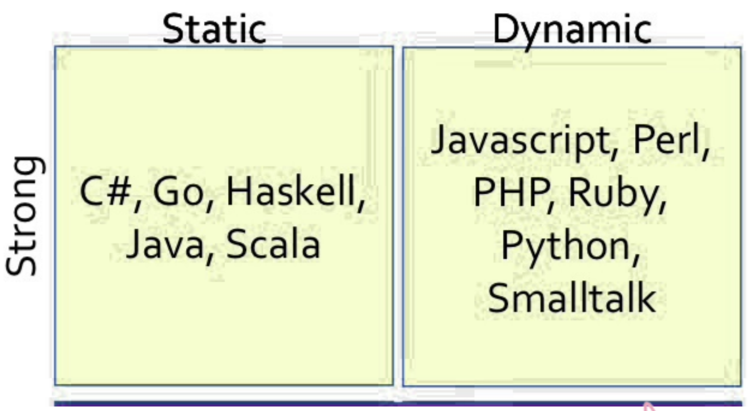


Answer:

A variable can't be assigned to a value of a new type. So n's type is fixed as a String - this is Static Typing! That means that n has a fixed type – thus, this language must use type inference! This is Kotlin!

# Type Checking Approaches

## Compile-time vs. Run-time



| | Static | Dynamic |
|---|---|---|
| Strong | C#, Go, Haskell, Java, Scala | Javascript, Perl, PHP, Ruby, Python, Smalltalk |

Strictness

| | Static | Dynamic |
|---|---|---|
| Strong | Strong type checking<br>The language's type system guarantees that all operations are only invoked on objects/values of appropriate types | |
| Weak | Assembly Language, C, C++ | |

# What is a Strongly-typed Language?

A strongly-typed language ensures that we will NEVER have undefined behavior at run time due to type-related issues.

— unlike weak

In a strongly-typed language , there is no possibility of an unchecked runtime type error.

(C, C++, assembly)

These are the minimum requirements to be strongly typed:

## The Language is Type-safe

The language is type-safe, meaning that it will prevent an operation on a variable X if X's type doesn't support that operation

```
int a;
Dog d;
a = 5 * d; // Prevented!
```

## The Language is Memory Safe

A memory-safe language prevents inappropriate memory accesses (e.g., out-of-bound array accesses, access to a dangling pointer)

```
int arr[5], *ptr;
cout << arr[10]; // Prevented!
cout << *ptr;    // Prevented!
```

always crashes

would work in C++

These can be enforced statically or dynamically.

even mostly strongly typed or C++ compiler

# Things We Expect in a Strongly Typed Language

Here are some of the techniques that languages use to implement strong typing:

checked by C++ compiler

Before an expression is evaluated, the compiler/interpreter validates that all of the operands used in the expression have compatible types.

```
y = Dog("Koda")
x = 5 + y
```

All conversions/casts between different types are checked and if the types are incompatible, then an exception will be generated.

```
y = Dog("Koda")
x = (int)y
```

Pointers are either set to null or assigned to point at a valid object at creation.

```
Dog *x
print(x)  // NULL!
```

Accesses to arrays are bounds checked; pointer arithmetic is bounds-checked.

```
int x[5]
print(x[100])
```

The language ensures objects can't be used after they are destroyed.

```
delete d;
d->bark();
```

General principle: Prevent operations on incompatible types or invalid memory.

instead of undefined

# Memory Safety and Strong Typing?

behavior

in C++

Challenge: Why must a language be memory-safe to be considered strongly-typed?

Here's a hint.

```
// C++
int arr[3] = {10,20,30};
float salary = 120000.50;

cout << arr[3];
```

# Memory Safety and Strong Typing?

Challenge: Why must a language be memory-safe to be considered strongly-typed?

Here's a hint.

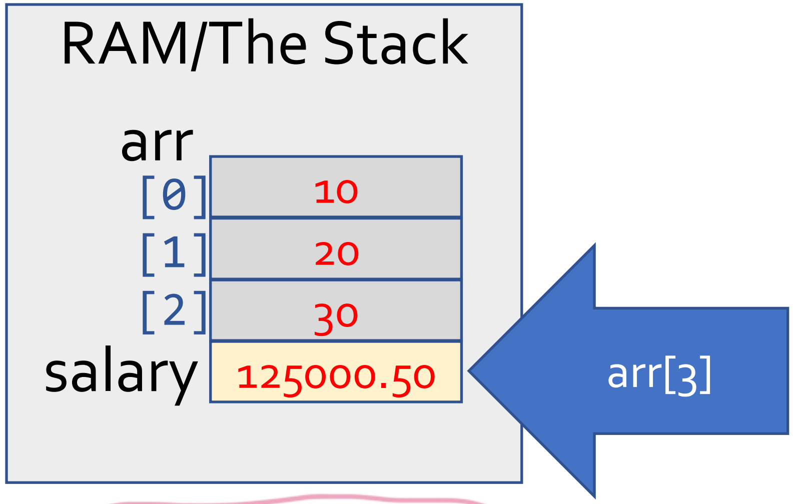
But it's a floating point variable!

```
// C++
int arr[3] = {10,20,30};
float salary = 120000.50;

cout << arr[3];
```

arr[0] arr[1] arr[2]

accessing salary via arr[3]

This accesses the salary variable as if it were an integer!



Answer: If a language is not memory safe, you might access a value (like salary) using the wrong type (int instead of float)!

Here's another example!

```
// Answer: Accessing a dangling pointer!
float *ptr = new float[100];
delete [] ptr;
cout << ptr[0]; // is that still a float?!
```

wrong type

also accessing value that isn't there

# Strongly Typed Languages: Checked ~~Cats~~ Casts

cast runtime

A checked cast is a type-cast that results in an exception/error if the cast is illegal!

strong

```
// Strongly-typed Java has "checked" casts
public void petAnimal(Animal a) {
  a.pet(); // Pet the animal

  Dog d = (Dog)a;
  d.wagTail();
}

...

public void takeCareOfCats() {
  Cat c = new Cat("Meowmer");
  petAnimal(c);
}
```

java.lang.ClassCastException: class Cat cannot be cast to class Dog

#1: Strongly-typed Java ensures we never succeed with an incomptible cast!

weak

```
// Unlike C++'s "unchecked" casts
void petAnimal(Animal *a) {
  a->pet(); // Pet the animal

  Dog* d = (Dog *)a;
  d->wagTail();
}

...

void takeCareOfCats() {
  Cat c("Meowmer");
  petAnimal(&c);
}
```

#2: This code runs even though were dealing with a Cat, not a Dog!

#3: At this point, anything could happen!

not a dynamic cast, so not checked

# Why Should We Prefer Strongly Typed Languages?

# So Why Do People Still Use Weakly Typed Languages?

# Why Should We Prefer Strongly Typed Languages?

Earlier detection and fixing of bugs/errors

Dramatically-reduced software vulnerabilities (less hacking)

buffer overflows

etc.

# So Why Do People Still Use Weakly Typed Languages?

~~Performance and legacy.~~

performance and legacy

# The Definition of Strong Typing is Strongly Disputed ☺

Many academics argue for a broader definition of strong typing, e.g.:

All conversions between different types must be explicit

The language has to have explicit type annotations for each variable

The type of each variable can be determined at compilation time

etc…

And some strongly-typed languages even have these features.

But while these items may make a language's type system stricter, they ultimately don't impact the language's type safety or its memory safety.

So we won't use them for our definition.

not true

haskell
but offers
string casting
allow
implicit
casts

not true
for
python,
but
checked
at
runtime

# Type Checking Approaches

## Compile-time vs. Run-time

Strictness

| | Static | Dynamic |
|---|---|---|
| Strong | C#, Go, Haskell, Java, Scala | Javascript, Perl, PHP, Ruby, Python, Smalltalk |
| Weak | Weak type checking<br>The language's type system does NOT guarantee that all operations are invoked on objects/values of appropriate types | |

Assembly

C, C++

# What is a Weakly Typed Language?

Here are some attributes associated with weakly-typed languages:

## They are not Type-safe

The language may not detect or prevent operations on data types that don't support those operations

```
Lion leo;
leo.quack(); // ???
```

but compiler might

## They are not Memory Safe

Programs may access memory outside of array bounds or via dangling pointers

```
int arr[3];
cout << arr[9];
```

```
int *ptr;
cout << *ptr;
```

# Weak Typing and Undefined Behavior

In a strongly typed language, we know that all operations on variables will either succeed or generate an explicit type exception at runtime (in dynamically-typed languages).

runtime type checking

But in weakly-typed languages, we can have undefined behavior at runtime!

downcasting

```
// C++ int → Nerd example w/undefined behavior!
class Nerd {
public:
  Nerd(string name, int IQ) { ...}
  int get_iq() { return iq_; }
  ...
};

int main() {
  int a = 10;
  Nerd *n = reinterpret_cast<Nerd *>(&a);
  cout << n->get_iq(); // ?? What happens?!?!?
}
```

This reinterprets our integer as if it were a Nerd object!

Then tries to call the get_iq() method… of course it crashes!

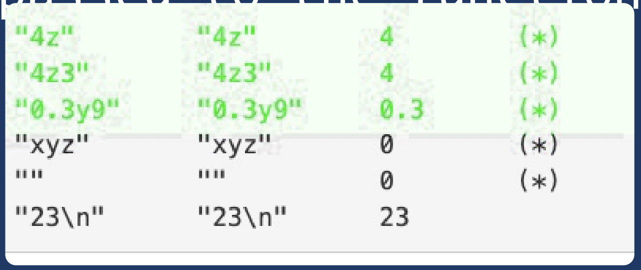
re-interpret "normal" cast between variables

reinterpret vs. dynamic vs. static cast

done at compile time conversion

# Classify That Language: Type Checking

```
# Defines a function called ComputeSum
# In this language, @_ is an array that holds
# all arguments passed to the function
```



```
"4z"        "4z"        4       (*)
"4z3"       "4z3"       4       (*)
"0.3y9"     "0.3y9"     0.3     (*)
"xyz"       "xyz"       0       (*)
""          ""          0       (*)
"23\n"      "23\n"      23
```

```
sub ComputeSum {
   $sum = 0;

   foreach $item (@_) {     # loop thru args
      $sum += $item;
   }

   print("Sum of inputs: $sum\n")
}

# Function call
ComputeSum(10, "90", "cat");
```

90

150

strongly typed

implicitly converted but defined

We've run this code a million times, and each time it prints:

```
Sum of inputs: 100
```

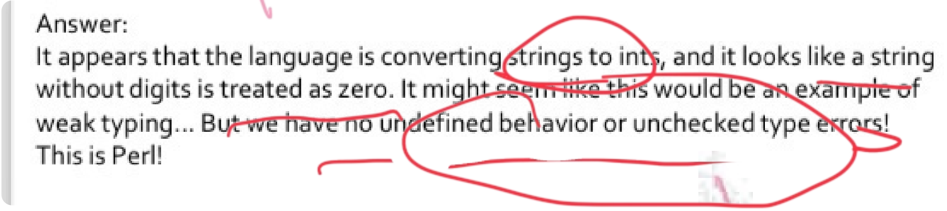
not allowed in strong typing

Is this language likely strongly or weakly typed?

would crash or be undefined

X

10 + 90 + 0 'string' treated as int



Answer:
It appears that the language is converting strings to ints, and it looks like a string without digits is treated as zero. It might seem like this would be an example of weak typing… But we have no undefined behavior or unchecked type errors! This is Perl!

# Classify That Language: Type Checking

```
fun processArgBasedOnType(x: Any) {
  when (x) {
    is Int -> print(x)
    is String -> print(x.length)
    is IntArray -> print(x.sum())
    else -> print((x as Dog).bark())
  }
}

fun main() {
  var x = Person("Carey","Nachenflopper");
  processArgBasedOnType(x)
}
```

Run-time: class Person cannot be cast to class Dog

Consider the following program which generates a runtime error:

Is this language strongly or weakly typed?

From this code, is it possible to determine if this language is statically or dynamically typed?

# Classify That Language: Type Checking

type introspection

#1: In this language, the "Any" type is a supertype of all other types.

but objects don't have types in dynamic

```
fun processArgBedOnType(x: Any) {
  when (x) {
    is Int -> print(x)
    is String -> print(x.length)
    is IntArray -> print(x.sum())
    else -> print((x as Dog).bark())
  }
}

fun main() {
  var x = Person("Carey","Nachenflopper");
  processArgBasedOnType(x)
}
```

The language is preventing invalid casting (at runtime): strongly-typed!

Run-time: class Person cannot be cast to class Dog

casts is looking at var and interpreting it as diff type

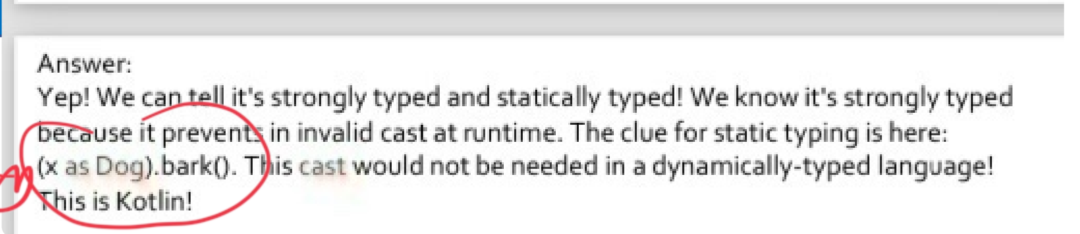
#2: Every other type is compatible with it – so we can pass in a Person, an Int, a Dog, etc.

don't need casts in dynamic, just x.bark() → duck typing

Consider the following program which generates a runtime error:

Is this language strongly or weakly typed?

From this code, is it possible to determine if this language is statically or dynamically typed?



Answer:

Yep! We can tell it's strongly typed and statically typed! We know it's strongly typed because it prevents in invalid cast at runtime. The clue for static typing is here: (x as Dog).bark(). This cast would not be needed in a dynamically-typed language! This is Kotlin!

# Static vs Dynamic, Strong vs Weak? What's Best?

The trend – in industry – is toward more strongly-typed languages with static type checking.

Facebook has developed Hack, a strongly and statically typed version of PHP (for backend web apps)

Facebook has developed Flow, a static type checker for JavaScript

Microsoft has developed TypeScript, a strongly and gradually typed version of JavaScript.

In fact, just about the only weakly typed languages left are C and C++.

# Language Of Th

## History

Dynamically typed

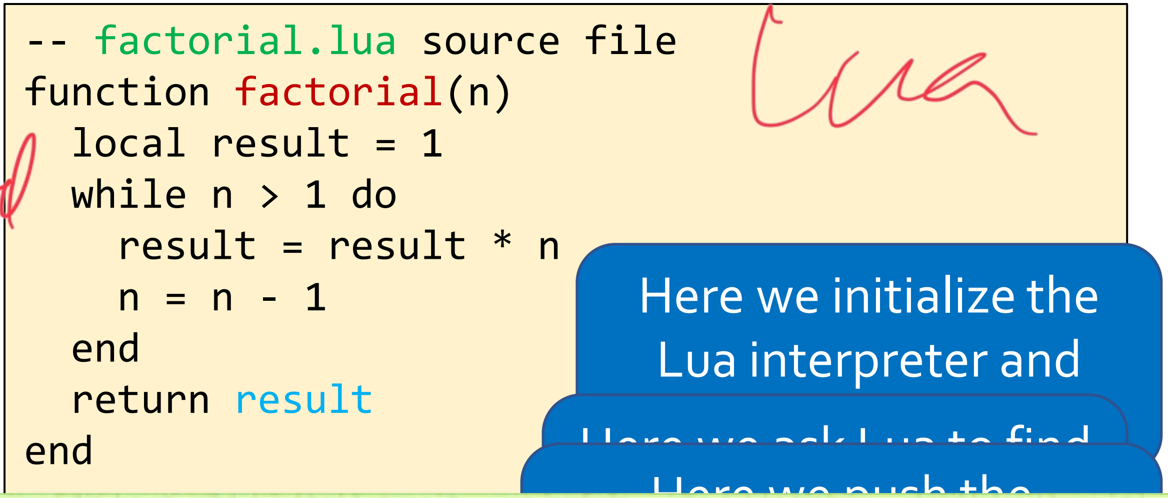
Lua was created in 1993 by three members of the Computer Graphics Technology Group at the Pontifical Catholic University of Rio de Janeiro.

## Overview

Lua is an interpreted language that comes as a library that can be integrated into other applications to let you add scripting to them.

## Unique Aspects

You can give your users the ability to customize your app by writing their own Lua scripts – e.g. in World of Warcraft, to automate in-game actions for the user.



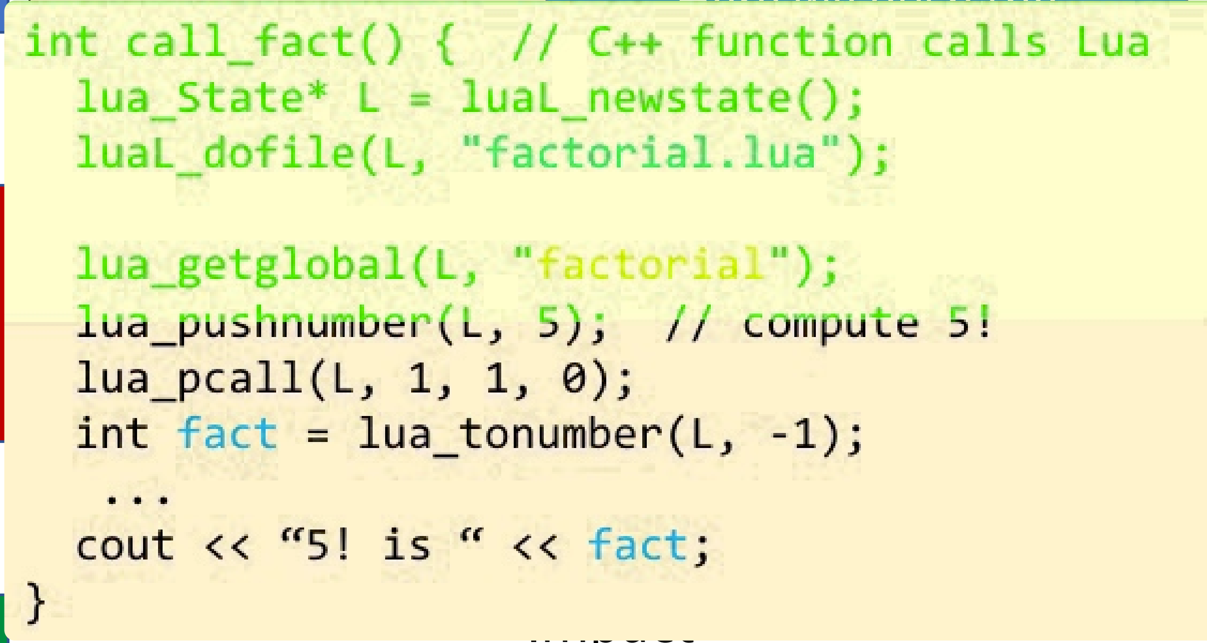
```lua
-- factorial.lua source file
function factorial(n)
  local result = 1
  while n > 1 do
    result = result * n
    n = n - 1
  end
  return result
end
```

Lua

Here we initialize the Lua interpreter and

Here we ask Lua to find

Here we push the



```cpp
int call_fact() {  // C++ function calls Lua
  lua_State* L = luaL_newstate();
  luaL_dofile(L, "factorial.lua");

  lua_getglobal(L, "factorial");
  lua_pushnumber(L, 5);  // compute 5!
  lua_pcall(L, 1, 1, 0);
  int fact = lua_tonumber(L, -1);
   ...
  cout << "5! is " << fact;
}
```

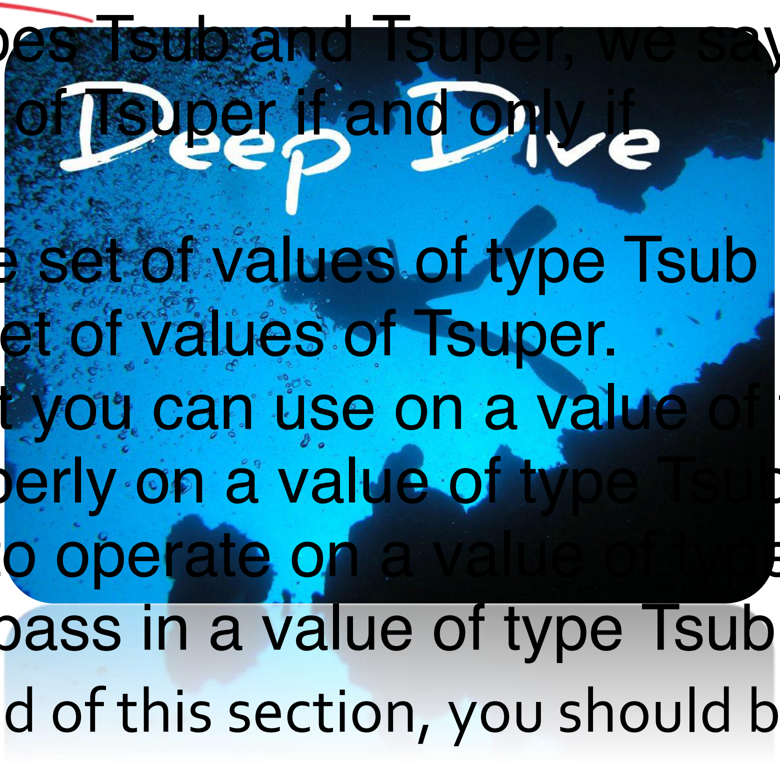
## Impact

Lua is used across diverse systems such as embedded platforms, antivirus engines,  databases (e.g., Redis) , etc.

# Type Conversion & Casting

So in language theory, we say that float is a subtype of double, or alternatively that double is a supertype of float.

More formally, given two types Tsub and Tsuper, we say that Tsub is a subtype of Tsuper if and only if,



every element belonging to the set of values of type Tsub is also a member of the set of values of Tsuper.

All operations (eg +, -, *, /) that you can use on a value of type Tsuper must also work properly on a value of type Tsub.

i.e., If I have code designed to operate on a value of type Tsuper, it must also work if I pass in a value of type Tsub.

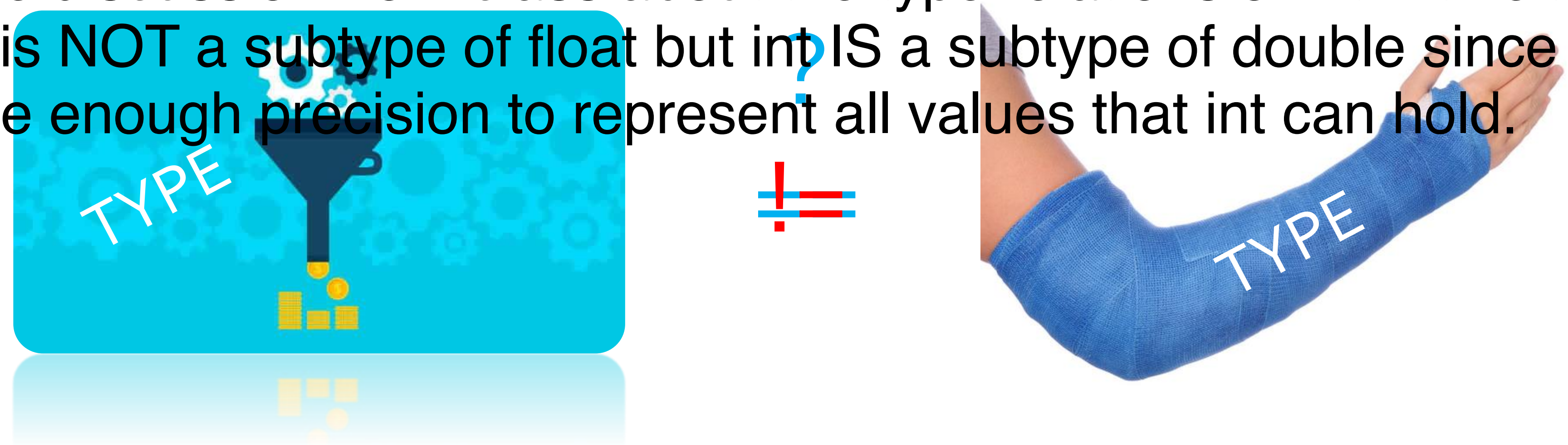
By the end of this section, you should be able to:

Take a new language and figure out the rules it uses to convert between different data types.

Understand the implications of its conversion approach so you can properly convert between different types in that language.

# Type Conversions and Type Casts

To clear up the discussion from class about the type relations of int with either float or double: int is NOT a subtype of float but int IS a subtype of double since doubles have enough precision to represent all values that int can hold.



Type conversion and type casting are used when we want to perform an operation on a value of type A, but the operation requires a value of type B, e.g.

*we want to pass an int value to a function that accepts a float value* conversion

*we want to add a long value to a double value in an expression* ← conversion

*we want to pass a Student object to a function that accepts a Person object (assuming Student is derived from Person)* ← Cast

# Two Options: Type Conversions and Type Casts

## Type Conversion

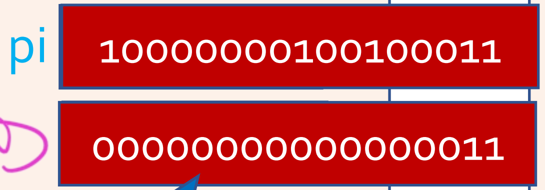
A conversion takes a value of type A and generates a whole new value (occupying new storage, with a different bit encoding) of type B.

Type conversions are typically used to convert between primitives (e.g. float → int).

```
// Conversion example

int main() {
  float pi = 3.141;
  cout << (int)pi; // 3
}
```

pi 3.14

3



The program performs a conversion, and generates a temporary new value of a different type in the process.

The converted value occupies distinct storage and has a different bit representation than the original value.

## Type Casting

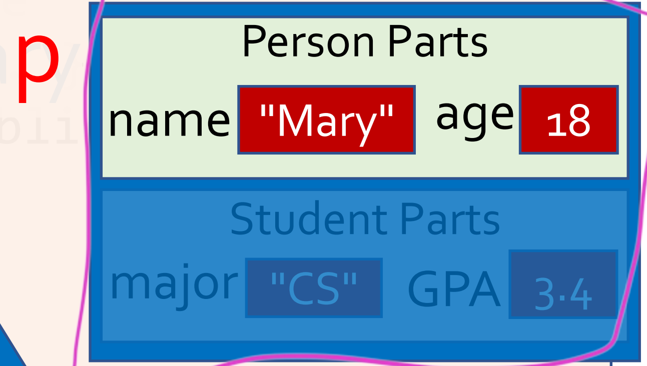
A cast takes a value of type A and views it as if it were value of type B – no conversion takes place! No new value is created!

Type casts are *typically* used with objects.

This cast lets us refer to our original Student object, but interpret it as if it were just a Person.

```
int main() {
  Student mary;

  ...
  Person &p = (Person&)mary;
  cout << "Hi " << p.name();
}
```

upcast



# Two Options: Type Conversions and Type Casts

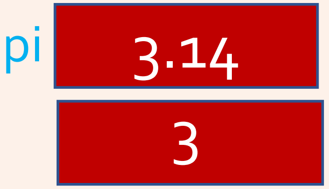
## Type Conversion

static and dynamic

A conversion takes a value of type A and generates a whole new value (occupying new storage, with a different bit encoding) of type B.

Type conversions are typically used to convert between primitives (e.g. float → int).

```
// Conversion example


int main() {
  float pi = 3.141;
  cout << (int)pi; // 3
}
```


pi

3.14

3

## Type Casting

only static

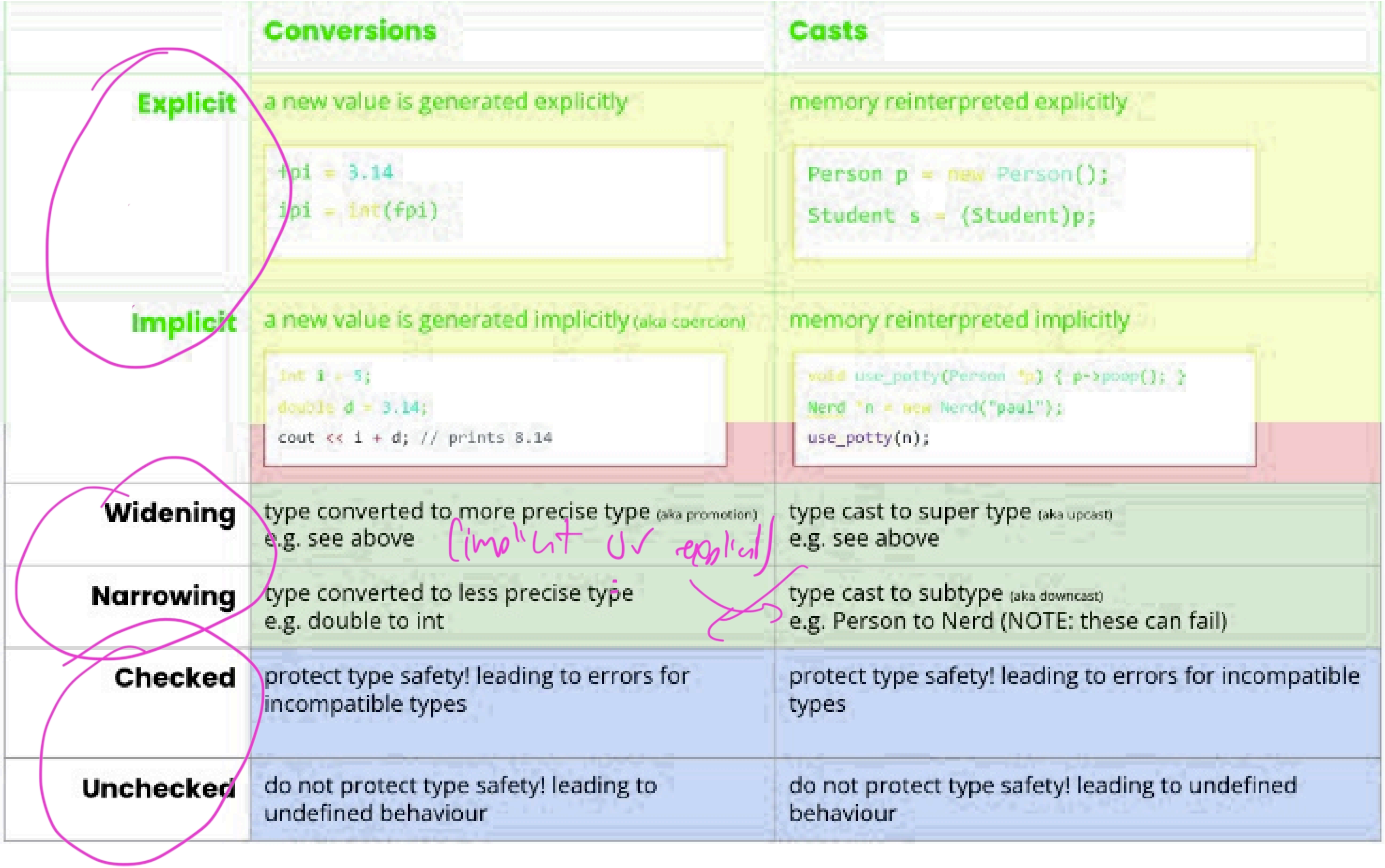
A cast takes a value of type A and views it as if it were value of type B – no conversion takes place! No new value is created!

Type casts are *typically* used with objects.

```
// Another casting example; treat an
// int as if it's an unsigned int!


int main() {
  int val = -42;

  cout << (unsigned int)val;
       // prints 4294967254
}
```

This refers to our original integer, but "interprets" its bits as if they represented an unsigned int.

# Casts and Conversions: Three Categories



| | Conversions | Casts |
|---|---|---|
| Explicit | a new value is generated explicitly<br>`fpi = 3.14`<br>`ipi = int(fpi)` | memory reinterpreted explicitly<br>`Person p = new Person();`<br>`Student s = (Student)p;` |
| Implicit | a new value is generated implicitly (aka coercion)<br>`int i = 5;`<br>`double d = 3.14;`<br>`cout << i + d; // prints 8.14` | memory reinterpreted implicitly<br>`void use_potty(Person *p) { p->poop(); }`<br>`Nerd *n = new Nerd("paul");`<br>`use_potty(n);` |
| Widening | type converted to more precise type (aka promotion)<br>e.g. see above (implicit or explicit) | type cast to super type (aka upcast)<br>e.g. see above |
| Narrowing | type converted to less precise type<br>e.g. double to int | type cast to subtype (aka downcast)<br>e.g. Person to Nerd (NOTE: these can fail) |
| Checked | protect type safety! leading to errors for incompatible types | protect type safety! leading to errors for incompatible types |
| Unchecked | do not protect type safety! leading to undefined behaviour | do not protect type safety! leading to undefined behaviour |

# Conversions/Casts: Explicit vs. Implicit

not implicit

cast

Both conversions and casts can be explicit or implicit.

PARENTAL
ADVISORY
EXPLICIT CONTENT

An explicit conversion/explicit cast requires you to use explicit syntax to force the conversion/cast.

PARENTAL
ADVISORY
IMPLICIT CONTENT

An implicit conversion (aka coercion) or implicit cast is one which happens without explicit syntax.

```
// Explicit conversion
void foo(int i) { ... }

int main() {
  float f = 3.14;
  foo((int)f);
}
```

Here we use explicit syntax to indicate that we want to convert our float value to an int.

```
// Implicit conversion
void foo(float f) { ... }

int main() {
  int i = 42;
  foo(i);
}
```

Here we implicitly convert (aka *coerce*) the type of our integer into a type of float.

```
// Explicit cast
void feed_young(Animal *a) {
  if (a->has_fur()) {
    ((Mammal *)a)->produce_milk();
  }
}
```

downcast

```
// Implicit cast
void use_potty(Person *p) { p->poop(); }

int main() {
  Nerd *n = new Nerd("paul");
  use_potty(n);
}
```

Most implicit casts are "upcasts" - from a subclass to a superclass.

Here we implicitly upcast a Nerd object to a Person.

# Explicit Type Conversions

actually are casts

never implicit downcasts

Let's look at explicit conversions in different languages.

```
// Explicit C++ conversions
float fpi = 3.14;
int ipi = (int)fpi;                    // old way
int ipi2 = static_cast<int>(fpi); // new way
```

Ironically, while this creates a new value, and is technically a "conversion", C++ calls it a "cast".

but it's a conversion

dynamic cast

```
# Explicit Python conversions
fpi = 3.14
ipi = int(fpi)
```

reinterpret cast

```
// Explicit Rust conversion
let x = 65 as char;  // x is equal to 'A'
println!("'A' as an unsigned 16-bit int is : {}", x as u16);
```

```
-- Explicit JavaScript conversion
fpi = 3.14
ipi = parseInt(fpi) -- converts to int
```

# Explicit Type Casts

only for statically typed

Let's look at explicit casts in different languages.

```
// Explicit C++ cast
class Person { ... };
class Student: public Person { ... }

void make_em_study(Person *p) {
  Student *s = dynamic_cast<Student*>(p);
  if (s != nullptr)
    s->study();
}
```

```
// Explicit Java cast
class Person { ... }
class Student extends Person { ... }

...

void make_em_study(Person p) {
  // next line throws exception if p doesn't
  // refer to a Student object
  try {
    Student s = (Student)p;
    s.study();
  } catch (ClassCastException exception) {
    ...
  }
}
```

```
// Explicit Kotlin cast
open class Person(name: String) { ... }
class Student(name: String, gpa: Double):
              Person(name) { ... }

fun make_em_study(p : Person) {
  val s:Student? = p as Student?
  if (s != null)
    s.study()
}
```

# Why Do We Have Explicit Conversions and Casts?

When you use an explicit conversion or cast, you're telling the compiler to change what would be a compile time error into a runtime check.

```
class Person { ... }
class Student extends Person { ... }
class Professor extends Person { ... }

class Example
{
  public void do_your_thing(Professor q) {
    q.give_a_lecture();
  }
  public void process_person(Person p) {
    if (p.get_name() == "Carey") // p's name is Carey, so p
      do_your_thing(p);          // must refer to a Prof!
  }
  ...
}
```

#1: The programmer might know that this code will always work...

#2: But a statically typed compiler can't prove this, and so will generate a compiler error for this implicit conversion.

```
java.lang.ClassCastException: class Person cannot be cast to class Professor
```

can't do implicit downcast

for strong and weak CE

can't pass supertype to subtype

but all profs are people

some people are not profs

# Why Do We Have Explicit Conversions and Casts?

When you use an explicit conversion or cast, you're telling the compiler to change what would be a compile time error into a runtime check.

```
class Person { ... }
class Student extends Person { ... }
class Professor extends Person { ... }

class Example
{
  public void do_your_thing(Professor q) {
    q.give_a_lecture();
  }
  public void process_person(Person p) {
    if (p.get_name() == "Carey")
      do_your_thing((Professor)p);
  }
  public void boneheaded_function() {
    Student s = new Student("Carey");
    process_person(s);
  }
}
```

compiler check

since it might be undefined at runtime

but not necessarily fatal

ok

not ok

We won't have undefined behavior here...

So if some boneheaded coder did this...

We're telling the compiler: *"I know this conversion/cast looks dangerous, but trust me, I know what I'm doing."*

Of course, in a strongly typed language, the program will still perform a runtime check before allowing the cast operation!

java.lang.ClassCastException: class Student cannot be cast to class Professor

runtime check

# Implicit Conversions: Coercions and Promotions

## C++ Implicit Conversion Rules

If either operand is long double then
  Convert the other to long double

Else if either operand is double then
  Convert the other to double

Else if either operand is float then
  Convert the other to float

Else if either operand is unsigned long int then
  Convert the other to unsigned long int

Else if the operands are long int and unsigned int and long int can represent unsigned int then
  Convert the unsigned int to long int

…

coercion but not promotion

Most languages have a prioritized set of rules that govern implicit conversions (aka coercions) that are allowed to occur without warnings/errors.

For instance, here are the C/C++ rules for coercion during binary operations:

coercion + promotion

```
int i = 5;
double d = 3.14;
cout << i + d; // prints 8.14
```

So we say that i is "promoted" from int to double, since double can hold all int values (and more).

So C++ converts i to a double before the addition operation is performed.

In this expression, C++ picks the highest priority conversion rule that applies…

In PL lingo, a coercion that converts a narrow type into a wider type is called a type promotion.

In contrast, this is a coercion from int to bool – but *not* a type promotion.

```
int a = 5;
...
if (a) cout << "a is not 0";
```

char → int in C++ would be promotion

# Conversions: Widening vs. Narrowing

int is not a subset of float but is of double

Conversions can be widening or narrowing.

long int → short int

A widening conversion is one that converts a narrower type to a wider type, e.g.: int → long, float → double

```
// Widening conversion: short → int
void foo(int i) { ... }

int main() {
  short s = 42;
  foo(s);
}
```

int can represent integers between -2bil to 2bil, which incudes all short values!

short can represent integers between -32768 to 32767

Since a wider type can represent every value the narrower type can, widening conversions are "value-preserving" - the converted value is always the same.

promotion → widening implicit conversion (but not if explicit)

A narrowing conversion is one that converts from wider type to a narrower type, or between two unrelated types.

```
// Narrowing conversion: float → int
void foo(int i) { ... }

int main() {
  float f = 3.14;
  foo(f);
}
```

int → float as well

This is a narrowing conversion because float and int are unrelated types with different ranges of values!

Narrowing conversions are NOT value-preserving, meaning the converted/casted value *might be* different than the original!

or int → short

dynamic_cast better for downcast
static_cast for upcast

# Casts: Widening vs. Narrowing

→ to superclass   → to subclass

Casts can also be widening (an "upcast") or narrowing (a "downcast").

(casts only in statically HLL

(cannot do implicit narrowing cast

A widening cast, aka an "upcast", casts a subtype variable as its supertype, e.g.: Student → Person

also explicit if static_cast

A narrowing cast, aka a "downcast", is one that casts a supertype variable as one of its subtypes, e.g.: Person → Prof

compiler error

```
class Person { ... };
class Student: public Person { ... };

void chat_with(Person &p)
  { cout << "Hi " << p->get_name(); }

int main() {
  Student s("Tammy","CS");
  chat_with(s);
}
```

to a Person (supertype)

Because they're guaranteed to work, upcasts may be implicit too!

Here we upcast a Student (subtype)...

implicit upcast (widening cast)

```
class Person { ... };
class Prof: public Person { ... };

void do_thing(Person *p) {
 if (p->get_name() == "Carey") {
   Prof *q = dynamic_cast<Prof *>(p);
   q->give_lecture();
 }
 else p->talk();
}
```

explicit narrowing cast

Here we downcast a variable we're currently treating as a Person (supertype)...

to a Prof (subtype)

Enabling us to use the subtype's specific methods!

Upcasts are always safe because we know that every subtype object (e.g., Student) is guaranteed to have all of the properties of the supertype (e.g. Person).

Downcasts may fail if the actual object is not compatible with the downcasted type!

# Conversions/Casts: Checked or Unchecked

Conversions and Casts can be checked or unchecked.

In a strongly-typed language, every conversion/cast with the potential for an issue is checked for validity at runtime

```
// Checked conversion (Java)
class Organism { ... }
class Alien extends Organism { ... }
class Dog extends Organism { ... }

...

public void play_time(Organism o) {
  Dog d = (Dog)o;
  d.play_fetch();
}

...

Alien a = new Alien(...);
play_time(a);
```

explicit narrowing cast

java.lang.ClassCastException: class Alien cannot be cast to class Dog

checked cast

implicit upcast (widen)

In a weakly-typed language, some invalid conversions/casts may not be checked (leading to undefined behavior)

```
// Unchecked conversion (C++)
class Organism { ... }
class Alien: public Organism { ... }
class Dog: public Organism { ... }

void play_time(Organism* o) {
  Dog* d = (Dog *)o;  // No error generated!
  d->play_fetch();    // Undefined behavior!
}

...

Alien *a = new Alien(...);
play_time(a);
```

unchecked cast

But dynamic casts are still checked casts

# Classify That Language: Casting & Conversion

```
function print(q) { /* ... */ }

y = '5' + 3;
print(y)
y = '5' - 3;
print(y)

print('5' + 3 - 3);
```

→ string concat → coerces int to char string

→ subtraction, coerces char string to int

int 53

string 3

The program to the left prints:

```
53
2
```

→ 50 → "53" - 3 = 50

Question #1: Does this language support coercion?

Yes

coercion

Question #2: Is this language statically or dynamically typed?

Yes

string 53 and int 2

Question #3: Assuming expressions are evaluated from left-to-right, what does this added last line print?

# Classify That Language: Casting & Conversion

Thanks to Matthew Wang for inspiring this problem!

```
function print(q) { / ..

y = '5' + 3;
print(y)
y = '5' - 3;
print(y)

print('5' + 3 - 3);
```

Q1#: Yes! The language coerces 3 into the string '3' when we use the + operator:

'5' + 3 → '5' + '3' → '53'

Q2: We first assign variable y to a string here...

Q1: And... the language coerces '5' into a number 5 when we use the - operator:
'5' - 3 → 5 - 3 → 2

Q2: and then assign y to a number here...

So this must be a dynamically typed language!

Finally, if we evaluate from left to right, this:

1. Concatenates '5' and '3' to get '53'
2. Subtracts 3 from 53, to get 50

This is JavaScript!

The program to the left prints:

```
53
2
50
```

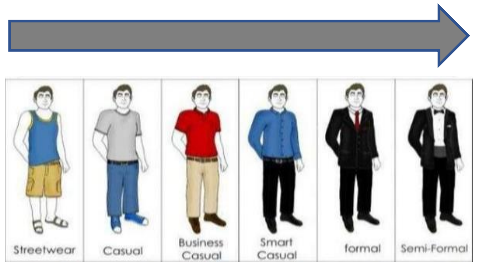
Question #1: Does this language support coercion?

Question #2: Is this language statically or dynamically typed?

Question #3: Assuming expressions are evaluated from left-to-right, what does this added last line print?

no implicit
conversions in
Haskell

# Types – A Final Thought



Type systems empower you to formalize a problem's structure into (user-defined) types.

This allows the compiler to verify that structure, enabling you to write more robust software.

checked/unchecked
casts/conversions

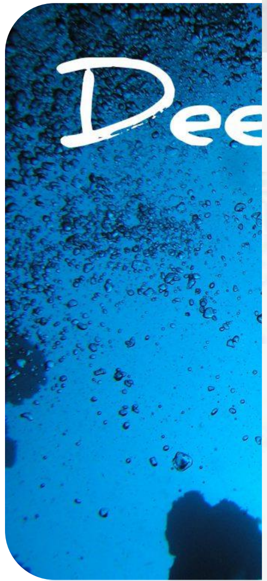
— int is subtype of
const int

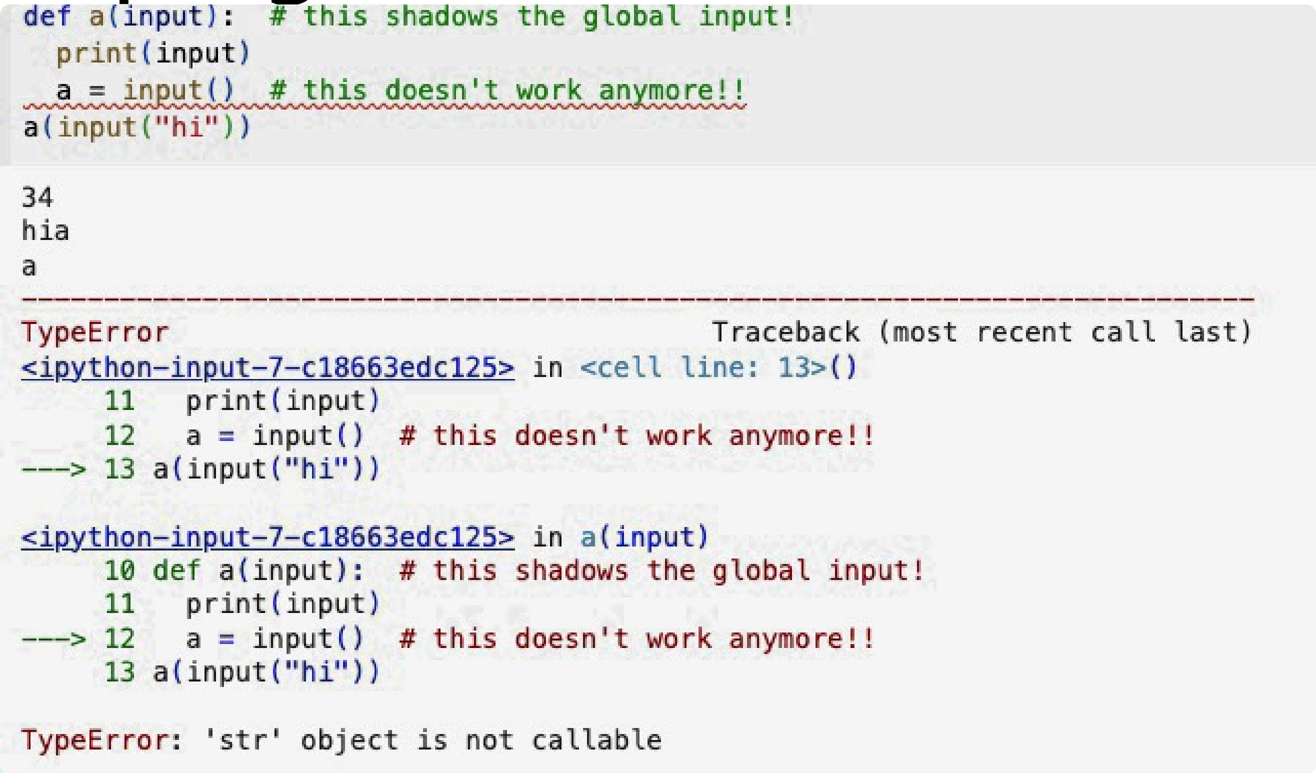
# Scoping

Python doesn't scope w/ respect to blocks:

```
def a():
  if True:
    x=5
  print(x)
```

← ok!





```
def a(input):  # this shadows the global input!
  print(input)
  a = input()  # this doesn't work anymore!!
a(input("hi"))
```

```
34
hia
a
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-7-c18663edc125> in <cell line: 13>()
     11   print(input)
     12   a = input()  # this doesn't work anymore!!
---> 13 a(input("hi"))

<ipython-input-7-c18663edc125> in a(input)
     10 def a(input):  # this shadows the global input!
     11   print(input)
---> 12   a = input()  # this doesn't work anymore!!
     13 a(input("hi"))

TypeError: 'str' object is not callable
```

By the end of this section, you should be able to:

Take a new language and understand its approach to variable scoping.

Understand the implications of its scoping approach for the visibility/accessibility of variables in your program.

# Scoping

## What’s the big picture?

Every language has scoping rules, which govern the visibility of variables and functions within a program.

A variable is "in-scope" in a region of a program if it can be explicitly accessed by its name in that region.

```
void foo() {
    int x;
    cout << x; // Just fine, x is in foo’s scope!
}

void bar() {
    cout << x; // ERROR! x isn’t in bar’s scope!
}
```

Scoping rules tell us what variables are visible at every place in the code, and what to do when there are multiple variables of the same name.

# Some Definitions...

Definition

## Scope

The Scope of a variable is the range of a program's instructions where the variable is known

```
void foo() {
  int x;
  cout << x;
}
```

"The scope of the x variable is the function foo()."

Definition

## In -scope

We say that a variable is "in-scope" if it can be accessed by its name in a particular part of a program.

"The x variable is in-scope within the foo function because it is defined at the top of the function."

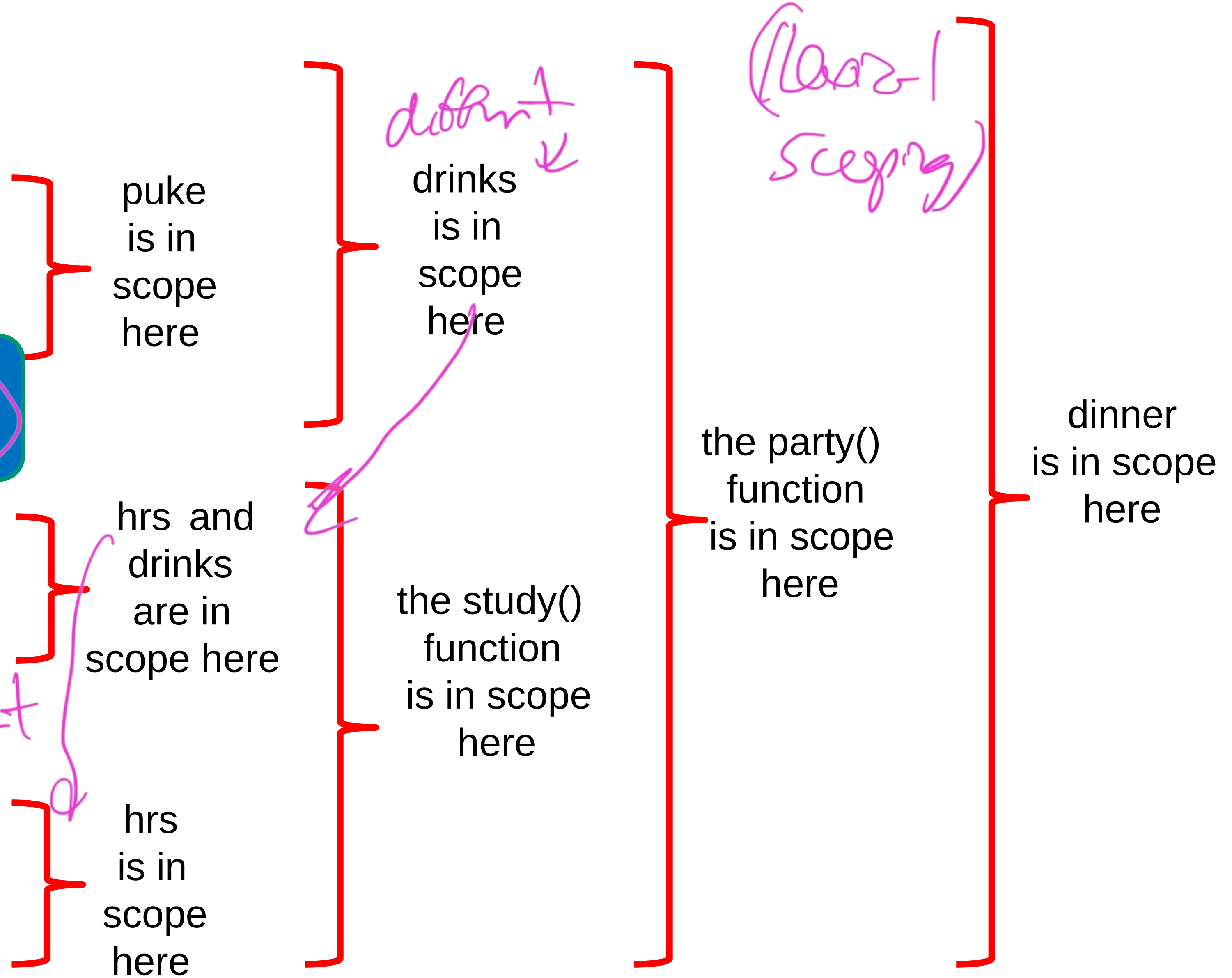
# A Simple C++ Scoping Example

```
string dinner = "burgers";

void party(int drinks) {
  cout << "Partay! w00t";
  if (drinks > 2) {
    bool puke = true;
    cout << "Puked " << dinner;
  }
}

void study(int hrs) {
  int drinks = 2;
  cout << "Study for " << hrs;
  party(drinks+1);
}

int main() {
  int hrs = 10;
  study(hrs-1);
}
```

#2: And that this hrs variable is totally different, and in-scope only in study().

#1: Note that this hrs variable is in-scope only in main().

different



puke is in scope here

hrs and drinks are in scope here

hrs is in scope here

different

drinks is in scope here

the study() function is in scope here

the party() function is in scope here

(Lexical scoping)

dinner is in scope here

# Scope changes as a program runs!

← need function or variable about to access

```
string dinner = "burgers";

void party(int drinks) {
  cout << "Partay! w00t";
  if (drinks > 2) {
    bool puke = true;
    cout << "Puked " << dinner;
  }
}

void study(int hrs) {
  int drinks = 2;
  cout << "Study for " << hrs;
  party(drinks+1);
}

int main() {
  int hrs = 10;
  study(hrs-1);
}
```

study is now out of scope, and drinks

← main and other hrs are not in scope anymore

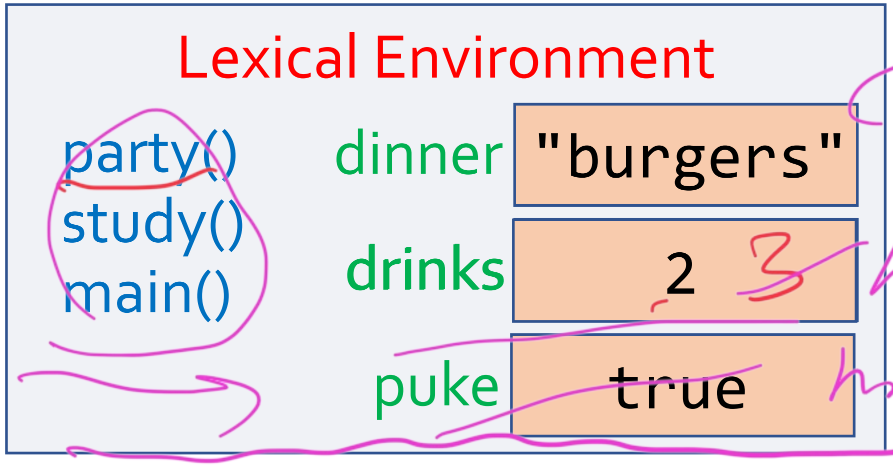
Another way to say that a variable is in scope is to say that it has an "active binding". "hrs is actively bound to storage which holds a value of 10."

Once a variable is in scope, it can be referred to by its name.

many things in scope here (but not variables inside functions)

Let's trace through this program and highlight actively in-scope variables in green and functions in blue!

The set of in-scope variables and functions at a particular point in a program is called its lexical environment.



Lexical Environment

| | | |
|---|---|---|
| party() | dinner | "burgers" |
| study() | drinks | 2 3 |
| main() | puke | true |

change over time

hrs - 10

hrs 9

The environment changes as variables come in or go out of scope.

# One More Definition...

Definition

## Lifetime (aka Extent)

Each variable also has a "lifetime" (from its creation to destruction).

A variable’s lifetime may include times when the variable is in scope, and times when it is not in scope (but still exists and can be accessed indirectly).

(ptr or obj ref)

```
void study(int how_long) {
  while (how_long-- > 0)
    cout << "Study!\n";
  cout << "Partay!\n";
}

int main() {
  int hrs = 10;
  study(hrs);
  cout << "I studied " << hrs <<
          " hours!";
}
```

#2: However, when we’re running the study() function, hrs is not in scope!

#1: The hrs variable has a lifetime that lasts from the start to the end of main()’s execution.

#3: But it still exists, and when study() returns, it will be back in scope!

```
def main():
  var = "I exist"
    ...
  del var     # no longer exists!
  print(var) # error!
```

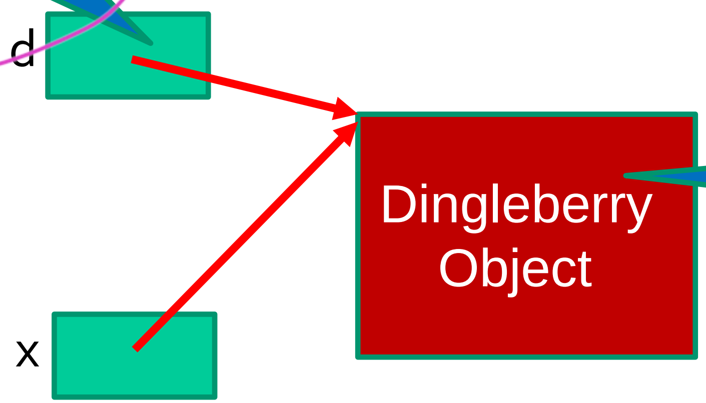
Some languages like Python allow you to explicitly control a variable's lifetime!

# Lifetimes… of Values

```
class Dingleberry:
 ...

def make_dingle():
  d = Dingleberry()
  return d

x = make_dingle()
if x.is_clinging():
  print("Wipey wipey")
```

#1: The d variable and the value it refers to are both "alive" while in the make_dingle() function.

#2: At this point, d's lifetime ends.



#3: But when we continue running in main(), the value d referred to is still alive, but now referred to by variable x!

Values also have lifetimes – and they're often independent of variables!

Let's see!

So while a variable's lifetime is limited to the execution of the function where it's defined…

A value may have a lifetime that extends indefinitely.

return variable
or pointer ←
or obj reference

# Lexical Scoping

Let's start by discussing Lexical Scoping, which is by far the dominant scoping approach.

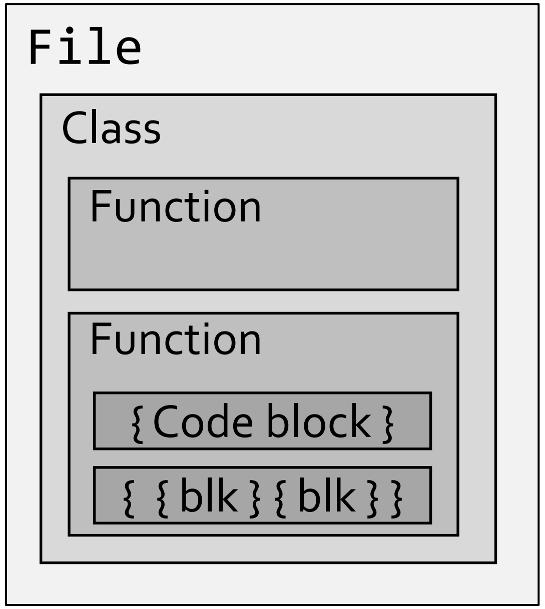
Definition

## Lexical (aka Static) Scoping

C++

All programs are comprised of a series of nested **contexts**: we have **files**, **classes** in those files, **functions** in those classes, **blocks** in those functions, blocks within blocks, etc.

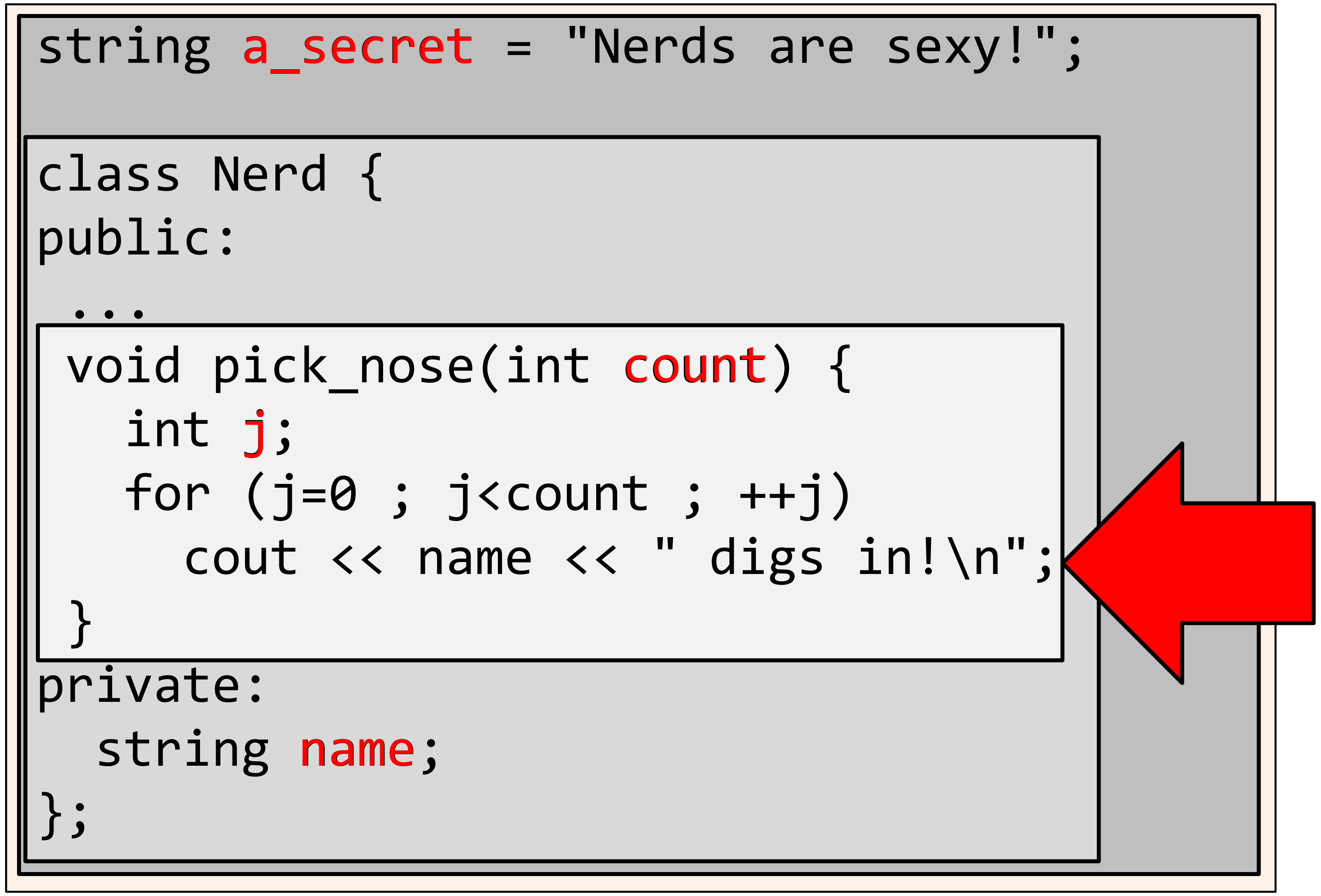
With lexical scoping, we determine all variables that are in scope at a position X in our code by looking at X's context first, then looking in successively larger enclosing contexts around x.



Virtually all modern languages use Lexical Scoping!

Why? The scoping rules are intuitive for coders, and scope can be computed unambiguously at compile time!

# Lexical Scoping (C++ Example)



```
string a_secret = "Nerds are sexy!";

class Nerd {
public:
  ...
 void pick_nose(int count) {
   int j;
   for (j=0 ; j<count ; ++j)
     cout << name << " digs in!\n";
 }
private:
  string name;
};
```

For instance, let's determine what variables are in-scope on this line right here...

Well, within our current function block, we have j and count in scope.

And within our enclosing class context, we see that the member variable name is also in scope!

Finally, when we expand to include our file context, we see that the global variable a_secret is also in scope!

So, in total, with lexical scoping, on this line j, count, name and a_secret are all in scope!

# Lexical Scoping (Python Example)

```
host = 'cindy'

def party():
  guest = 'chen'
  def use_hot_tub():
    drink = 'white claw'
    print(host,'and',guest,'are tubbin')
    print('and drinking', drink)
  use_hot_tub()
```

Python does scoping using the "LEGB" rule: Local, Enclosing, Global, and Built-in.

not if/else

**Local:**

First look in the current code block, function body or lambda expression.

**Enclosing:**

Then (if you have a nested function) look in the enclosing function that contains your function.

**Global:**

Then look at all of the top-level variables and functions.

**Built-in:**

Finally you're left with built-in python keywords, functions, etc.

In the local context, we discover drink.

Then in the enclosing context, we discover our guest.

Finally in the global context, we discover our host.

# What types of contexts do we consider for Lexical Scope?

## Expressions

A new variable is introduced as part of an expression, and its scope is limited to that expression.

```
let y = 5 in y*y
```

list comp

```
sum([x*x for x in
      range(10)])
```

Y and x have lexical scope only in their expressions

## Blocks

A new variable is introduced within a block, and its scope is limited to that block.

```
if (drinks > 2) {
 int puke = 5;
 …
}
```

```
if drinks > 2:
  puke = 5
  …
```

## Functions

A local variable or parameter is introduced within a function, and its scope is limited to that function.

```
void snore(int n) {
  int i = 0;
  while (i++ < n) …
}
```

# What types of contexts do we consider for Lexical Scope?

## Classes/Structs

A class can have member variables, whose scope is limited to that class.

```
class Dog {
public:
  void wash() {…}
 …
private:
  int num_fleas;
};
```

## Namespaces

Some languages have namespaces that also provide "cleaner" scoping.

```
namespace CONSTS {
  const float PI=3.14;
}

float area(float r) {
 return r*r*CONSTS.PI;
}
```

## Global

We can define global variables, whose scope is available to all functions in the program (or file).

```
# Global variable!
name = "Carey"

def who_am_i():
  print("I am ", name);
```

# Dynamic Scoping

## Dynamic Scoping

Definition

Similar to Lexical Scoping

In a language with dynamic scoping, when you reference a variable, the program tries to find it in the current block and its enclosing blocks…

If the variable can't be found, the program then searches the calling function for the variable. If it can't be fond there, it checks its calling function, etc.

The function That called it

Dynamic Scoping has a few holdovers (Logo, Emacs Lisp, Bash), but otherwise is DEAD!

```
func foo() {
  y++;
  print x, y
}

func bar() {
  int y = 32;
  foo();
}

func bletch() {
  int x = -1, y = 5;
  foo();
}

func main() {
  int x = 1000;
  bar();
  bletch();
}
```

```
1000  33
-1  6
```

same y

x and y are in scope

x and y are in scope

this x shadows main x

y goes away

y goes away again

x still 1000

# Classify That Language: Scoping

```
(setq a 100)  # sets a to 100

# prints the value of a
(defun print_value_of_a ()
  (print a))

# define local variable a, then
# call print_value_of_a
(let ((a -42))
  (print_value_of_a))
```

-42 shadows / 100

The following program outputs a value of -42

What does this imply about the type of scoping used by this language?

Dynamic Scope

dynamically bound

# Classify That Language: Scoping

```
(setq a 100)  # sets a to 100

# prints the value of a
(defun print_value_of_a ()
  (print a))

# define local variable a, then
# call print_value_of_a
(let ((a -42))
  (print_value_of_a))
```

global

→ if static, would print 100

The following program outputs a value of -42

for frame env (? no

What does this imply about the type of scoping used by this language?

global

checks current block then its calling function, then outer scope and its calling function, etc., so global is created last

Answer:
The value of a is taken from the calling function's local variable a, not the lexical scope
→ Dynamic Scoping! This is Emacs Lisp!

# Classify That Language: Lifetime

```
program main
  call foo()
  call foo()
  call foo()
end

subroutine foo()
    real :: a = 0
    a = a + 10
    write(*,*) "a = ", a
end
```

10

20

30

Bound once for 1st time

static variable

initialized on first call then maintains value

The following program outputs:

```
 a =    10.00000000
 a =    20.00000000
 a =    30.00000000
```

What does this imply about the lifetime of variables in this language?

What common problem-solving technique (starts with an "r") can we NOT use in this language?

recursion

# Classify That Language: Lifetime

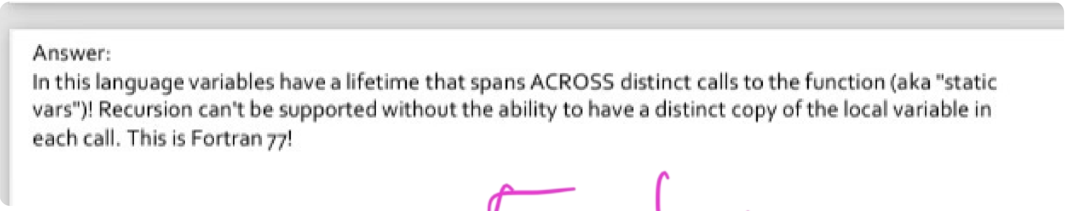
```
program main
  call foo()
  call foo()
  call foo()
end

subroutine foo()
    real :: a = 0
    a = a + 10
    write(*,*) "a = ", a
end
```

The following program outputs:

```
 a =      10.00000000
 a =      20.00000000
 a =      30.00000000
```

What does this imply about the lifetime of variables in this language?

What common problem-solving technique (starts with an "r") can we NOT use in this language?



Answer:
In this language variables have a lifetime that spans ACROSS distinct calls to the function (aka "static vars")! Recursion can't be supported without the ability to have a distinct copy of the local variable in each call. This is Fortran 77!

Fortran

Answer:
In this language variables have a lifetime that spans ACROSS distinct calls to the function (aka "static vars")! Recursion can't be supported without the ability to have a distinct copy of the local variable in each call. This is Fortran 77!

# Memory Safety

By the end of this section, you should be able to:

Take a new language and understand how it ensures safe access to memory to prevent bugs and hacking attacks.

Take a new language and understand how it reclaims the memory of "dead" objects as the program runs. — garbage collection

# Memory Safety

## What's the big picture?

Memory-safe languages prevent memory operations that could lead to undefined behaviors.

```
// Java does out-of-bounds checks on all array accesses
int[] array = new int[20];
int i = 400;
System.out.println(array[i]); // Java throws an exception!
```

Memory-unsafe languages allow memory operations that could lead to undefined behaviors.

```
// C++
int arr[3];
cout << arr[9]; // ????!?!
```

```
// Uninitialized pointer use
int *ptr;
cout << *ptr; // ???
```

An inordinate amount of bugs and hacking vulnerabilities are due to memory unsafety!

# Memory Unsafe Languages…

Weakly typed

Allow out-of-bound array indexes and unconstrained pointer arithmetic

```
int arr[10], *ptr = arr;
arr[-1] = 42;        // out-of-bound
cout << *(ptr + 100); // pointer arith'c
```

Allow casting values to incompatible types

```
int v;
Student *s = dynamic_cast<Student *>(&v);
s->study();
```

Allow use of uninitialized variables/pointers

```
int val, *ptr;     // both uninitialized
cout << val;       // could leak info!
*ptr = -10;        // corrupts memory
```

Allow use of dangling pointers to dead objects (programmer-controlled object destruction)

```
Student *s = new Student("Gerome");
delete s;   // student is no longer valid
s->study(); // ???
```

# Memory Safe Languages… (and: type safe) ↓ → strongly typed

~~Allow out-of-bound array indexes and unconstrained pointer arithmetic~~

Throw exceptions for out-of-bound array indexes;
Disallow pointer arithmetic

~~Allow casting values to incompatible types~~

Throw an exception or generate a compiler error for invalid casts

~~Allow use of uninitialized variables/pointers~~

Throw an exception or generate a compiler error if an uninitialized variable/pointer is used;
Hide explicit pointers altogether (e.g., Python)

~~Allow use of dangling pointers to dead objects (programmer-controlled object destruction)~~

Prohibit programmer-controlled object destruction
Ensure objects are only destroyed when *all* references to them disappear (Garbage Collection)

all [illegible] references but + strong memory safe ←

can still have memory leaks in python if garbage collector fails to reclaim

can be garbage collected or not explicitly freed

# Memory Safety and Memory Leaks

Shouldn't a language be considered unsafe if it can have memory leaks?

? not unsafe/ undefined

← just runs out of memory

Well, if our criteria for something to be "unsafe" is that it leads to undefined behaviors, then memory leaks don't count!

Why? Even languages with automated memory management (e.g., garbage collection) can sometimes run out of memory!

When this happens, the program is predictably terminated – there are no undefined behaviors.

So based on our criterion for memory safety, we will not require a language to prevent memory leaks.

# Strategies for Memory Leaks and Dangling Pointers

| Garbage Collection | Ownership Model |
|---|---|
| The language manages all memory de-allocation automatically | The compiler ensures objects get destroyed when their lifetime ends |
| C#, Go, Java, JavaScript<br>Python, Haskell, … | Rust<br>C++ (Smart Pointers) |

C3P

FUN FACT Garbage collection was pioneered in LISP in the early 60s.

# Garbage Collection

nondeterministic

Garbage Collection is the automatic reclamation of memory which was allocated by a program, but which is no longer referenced.

In a language with garbage collection the programmer does not explicitly control object destruction – the language does.

When a value or object on the heap is no longer referred to, the program (eventually) detects this at runtime and frees the memory associated with it.

What are the benefits? Let's see!

undefined behavior in C++

**Eliminates Memory Leaks** (free)

Ensures memory allocated for objects is freed once it's no longer needed

**Eliminates Dangling Pointers and Use of Dead Objects**

Prevents access to objects after they have been de-allocated

**Eliminates Double-free Bugs**

Eliminates inadvertent attempts to free memory more than once

**Eliminates Manual Memory Management**

Simplifies code by eliminating manual deletion of memory

# When Should Objects be Garbage Collected?

CHALLENGE! What criteria should be used to decide when to garbage collect an object?

# When Should Objects be Garbage Collected?

CHALLENGE! What criteria should be used to decide when to garbage collect an object?

Answer: A good rule of thumb: Garbage collect an object when there are no longer any references to that object.

No locals, no member variables, no globals, etc.

pointers/
obj references

```
public void do_some_work() {
    Nerd nerd = new Nerd("Jen");
    ...
} // nerd goes out of scope
```

```
public void do_some_work() {
    Nerd nerd = new Nerd("Jen");
    ...
    // we overwrite an obj ref
    nerd = new Nerd("Rick");
    // or
    nerd = null;
}
```

# Garbage Collection Approaches

mark and delete all dead

mark and move → slower

Let's talk about three of the main garbage collection approaches!

dead objects deleted at once → faster

## Mark and Sweep

Discover active objects by doing a traversal from all global, local and member variables that are obj references.

Free all objects that were not reached during discovery.

Go, Java, JavaScript

## Mark and Compact

Discover all active objects; move 'em into a new block of memory.

Throw away everything in the old block of memory (which holds only dead objects).

C#, Haskell

## Reference Counting

Each object keeps a count of the number of active object references that point at it.

When an object's count reaches zero, its memory is reclaimed.

Perl, Python, Swift

Bulk garbage collection occurs when free memory runs low – the program's execution is frozen temporarily while this happens!

Individual objects are garbage collected the moment their count reaches zero.

→ or garbage collection when idle

# Mark and Sweep Garbage Collection

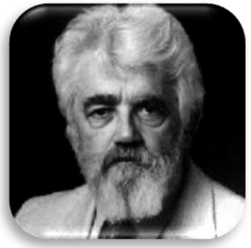
Mark and Sweep runs in two phases:

## A Mark Phase

The algorithm identifies all objects that are still referred to and thus considered to be in-use.

## A Sweep Phase

The algorithm scans all heap memory from start to finish, and frees all blocks not marked as being 'in-use.'



Mark and Sweep was invented by John McCarthy (inventor of LISP) in 1960

# Mark and Sweep: The Mark Phase

During the mark phase, our goal is to discover all active objects that are still being used.

We consider an object in-use (and its memory not reclaimable) if it meets one of two criteria:

## It is one of a key set of root objects

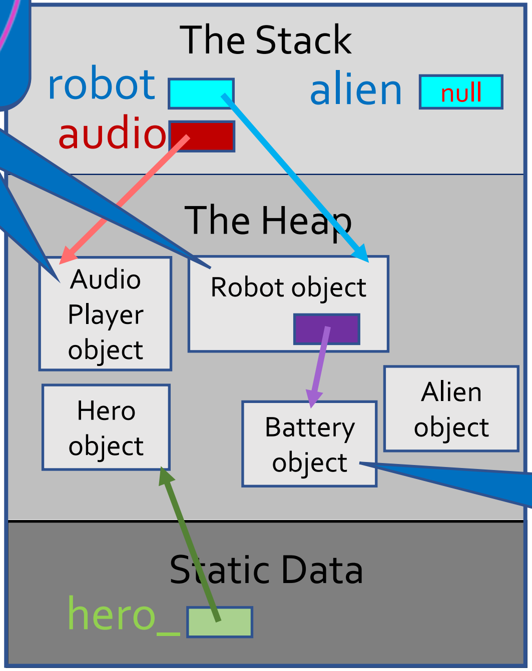
Root objects include global variables, local variables across all stack frames, and parameters on the call stack

## It is reachable from a root object

If an object can be transitively reached via one or more pointers/references from a root object (e.g.,robot object points to battery)

```
// Java
public class Game {
  public void play(AudioPlayer audio) {
    Robot robot = new Robot("Quark");
    Alien alien = new Alien();
    ...
    alien = null;
  }

  static Hero hero_ = new Hero();
}
```

By definition, all root objects are active and therefore should NOT be garbage collected.



And if an object is referred to by a root object, then it must be active too for the root object to function. And so on!

# Mark and Sweep: The Mark Phase

During the first part of the mark phase, the garbage collector identifies all root objects and adds their object references to a queue* for investigation.

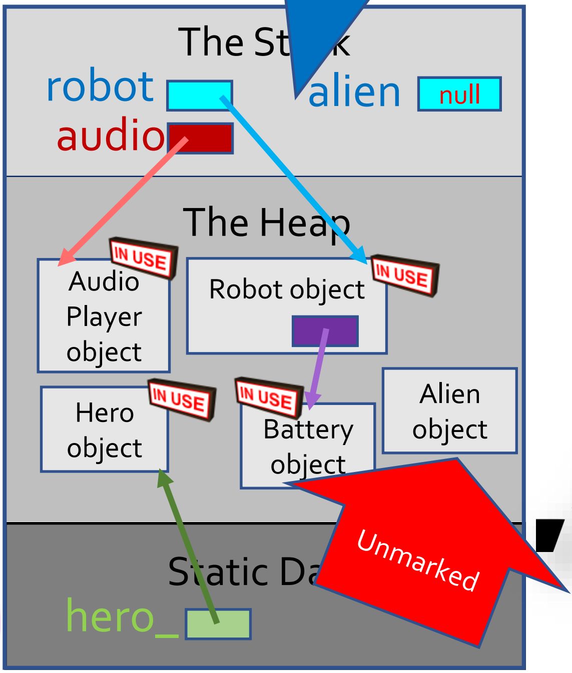
During the second part, the garbage collector uses the queue to breadth-first-search from the root objects and mark all reachable objects

Each object has a bit (hidden from the programmer) which is set by the GC to mark that it's still in-use.

```
# Pseudocode for the Mark algorithm
def mark():
  roots = get_all_root_objs()
  candidates = new Queue()
  for each obj_ref in roots:
    candidates.enqueue(obj_ref)

  while not candidates.empty():
    c = candidates.dequeue()
    for r in get_obj_refs_in_object(c):
      if not is_marked(r):
        mark_as_in_use(r)
        candidates.enqueue(r)
```

mark root objects

mark remaining objects



When we're done, all reachable objects have been marked.

All unmarked objects are not in use and can be disposed of!

How does the GC find unmarked variables?

# Mark and Sweep: The Sweep Phase

During the sweep phase, we traverse all memory blocks in the heap (each block holds a single object/value/array) and examine each object's in-use flag.

How do we traverse memory blocks?

Well, all memory blocks in the heap are linked together top-to-bottom in a linked list!

So to perform the sweep phase, we can simply follow the links from top-to-bottom.

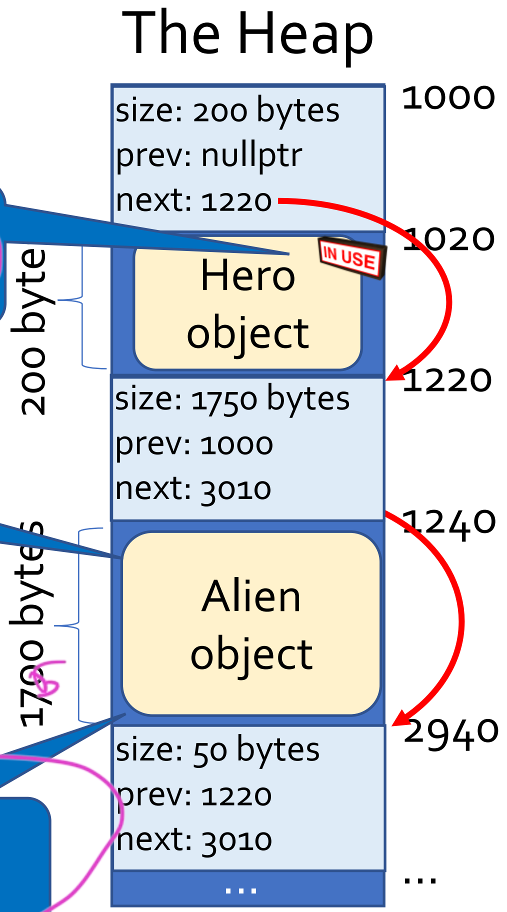
```
# Pseudocode for the Sweep algorithm
def sweep():
  p = pointer_to_first_block_in_heap()
  end = end_of_heap()
  while p < end:
    if is_object_in_block_in_use(p):
      reset_in_use(p)      # remove the mark, object lives
    else:
      free(p)              # free this block/object
    p = p.next
```

expensive to search

Our first object was marked as in-use, so we can keep it and just reset the in-use flag for next time.

Our second object was not marked as in-use, so we can free it.

Adjacent free blocks can then be coalesced into a single large block!

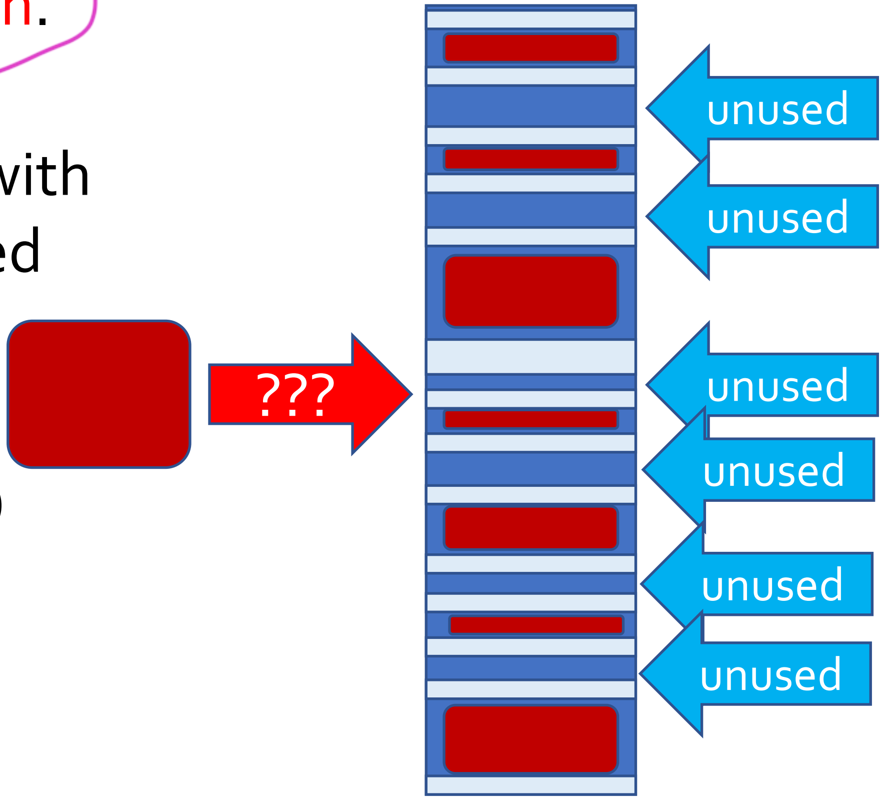


# Mark and Sweep: Memory Fragmentation

Mark and Sweep can result in memory fragmentation.

Fragmentation is when the heap becomes peppered with small, unused memory blocks where previously-freed objects used to be.

When this happens, it becomes slow (or impossible) to find free chunks of memory big enough to accommodate new object allocations.



So how might we deal with this? Let's see!

* Rather than using a queue or stack, the mark and sweep algorithm can use a clever pointer manipulation trick. But logically you can think of this as a breadth-first or depth-first traversal.

# Mark and Compact – A Twist on Mark and Sweep

In Mark and Compact GC, we perform our normal mark phase.

However, once we're done marking, we don't sweep away unmarked objects!

Instead, we compact all marked/in-use objects to a new contiguous block of memory.

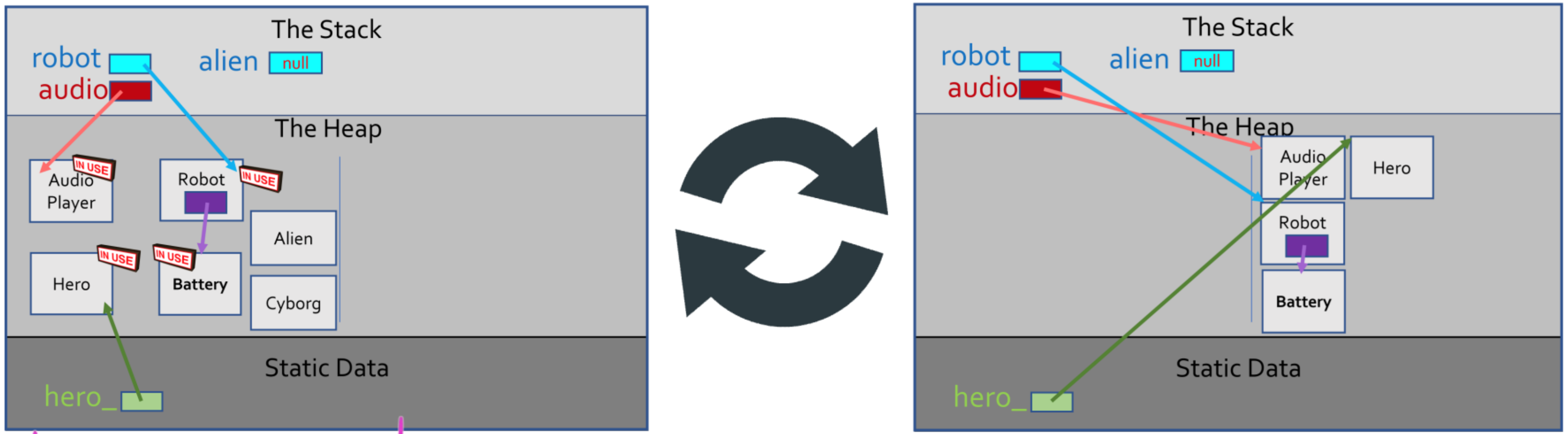
Then we adjust all pointers to the proper relocated-addresses.

Finally, our original block of memory is just treated as if it's empty and can be reused as a whole without dealing with any sweeping.

but also less fragmentation

can use only half of memory at a time

We alternate compaction back and forth between the two contiguous blocks.



also expensive to copy all objects but also expensive to sweep

# Garbage Collection and (Un)Predictability

With GC approaches, it's impossible to predict when (and if) a given object will actually be freed by the collector – collection only occurs when there's memory pressure.

non deterministic

Challenge: Why does it matter?

Guaranteed to be safe

– may miss objects but will not delete in-use objects

don't want to garbage collect at inopportune times

↳ such as when rocket is about to land

real-time systems often don't use GC and use explicit or Rust ownership model

# Garbage Collection and (Un)Predictability

but for "regular" objects, manual GC is slower

With GC approaches, it's impossible to predict when (and if) a given object will actually be freed by the collector – collection only occurs when there's memory pressure.

slows everything all threads

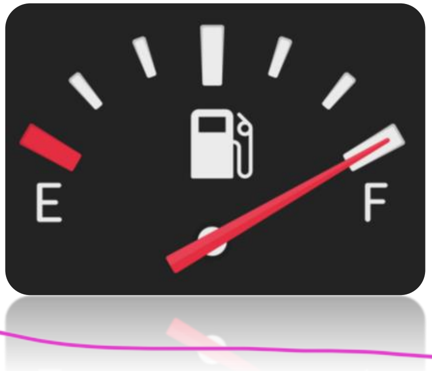
Academic Robot Says:

"□ -based languages, the programmer really needs to free other resources (e.g., files) manually and assume GC won't happen."

Challenge: Why does it matter?

Well, what if each object creates a large temporary file on the hard drive?

And what if there's plenty of RAM, so the collector doesn't run and get rid of unreachable objects (and their temp files) often?



You're going to run out of hard-drive space, long before you run out of RAM!

open and close files in Python

# Reference Counting-based Garbage Collection

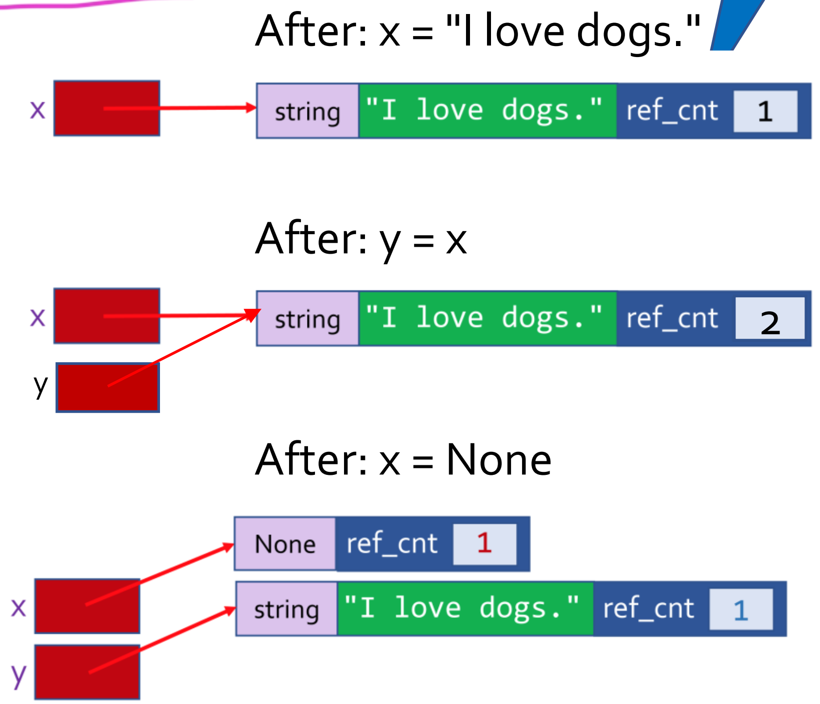
In reference counting GC, every object has a hidden count that tracks how many references there are to it.

A reference count is secretly stored with each object and array.

```
def foo():
  x = "I love dogs."
  y = x # y.ref_count += 1

  # x.ref_count -= 1
  x = None

  # locals go out of scope
  # y.ref_count -= 1
  # x.ref_count -= 1
```

The language secretly bumps up the count every time a new reference is created to the object.

The language secretly decrements the count every time a reference to it goes away.



Every time a new reference is created to an object, the language secretly increments the count.

Every time a reference to an object disappears, the language secretly decrements its count.

If an object's count reaches zero, the object is deleted.

x.ref_cnt is now None,

x still exists

(not del(x))

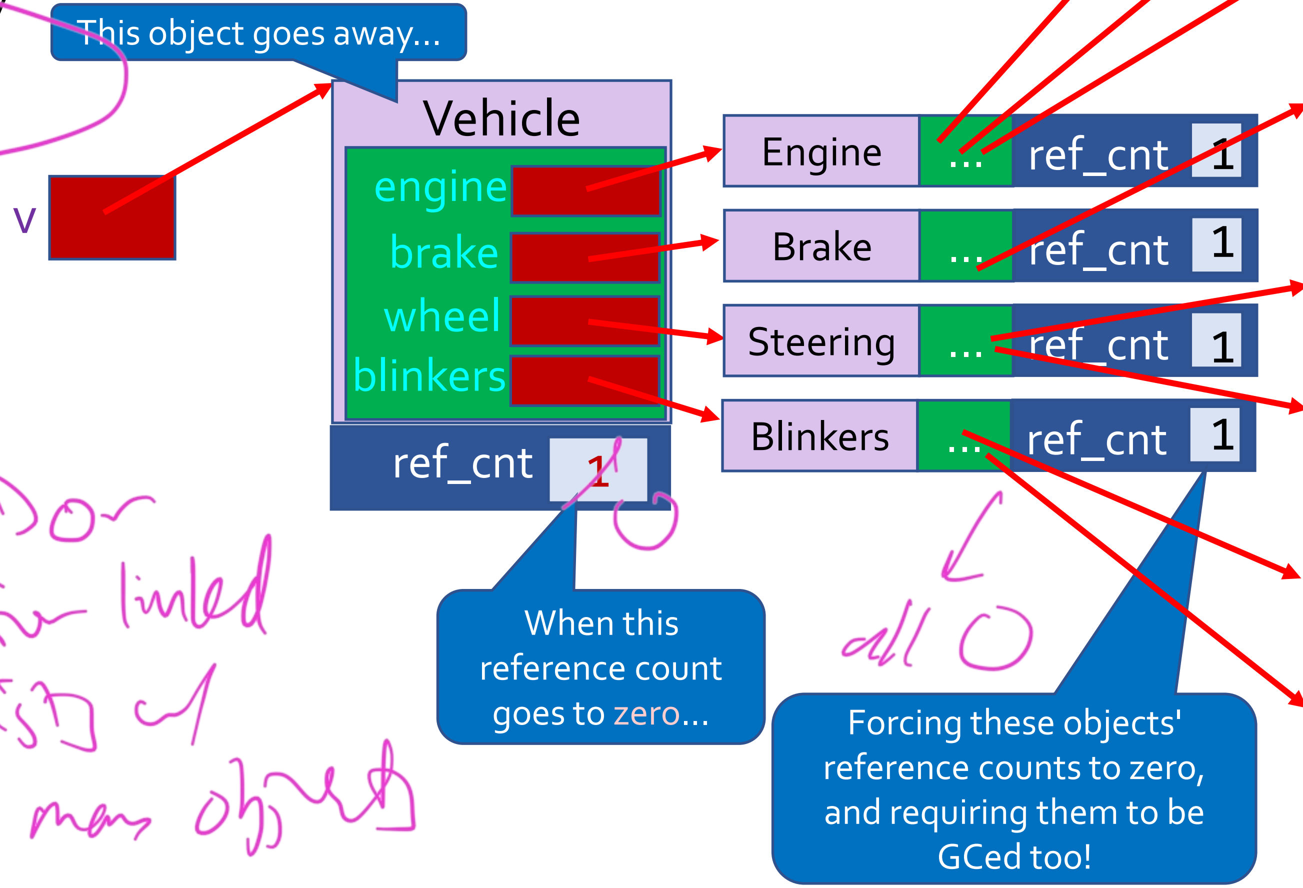
# Reference Counting-based Garbage Collection

When an object is destroyed (its reference count hits ZERO), all objects transitively referenced by that object must also have their reference counts decreased!

Because of this, removing a single reference can potentially lead to a cascade of objects being freed at once. SLOW!

```
class Vehicle:
  def __init__(self):
    self.engine = Engine()
    self.brake = Brake()
    self.wheel = Steering()
    self.blinkers = Blinkers()

def game():
  v = Vehicle()
  ...
  v = None
```

→ or for linked lists of many objects



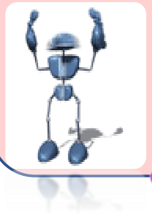
This object goes away...

When this reference count goes to zero...

Forcing these objects' reference counts to zero, and requiring them to be GCed too!

Challenge: How might we address this to speed things up in the average case?

# Reference Counting-based Garbage Co



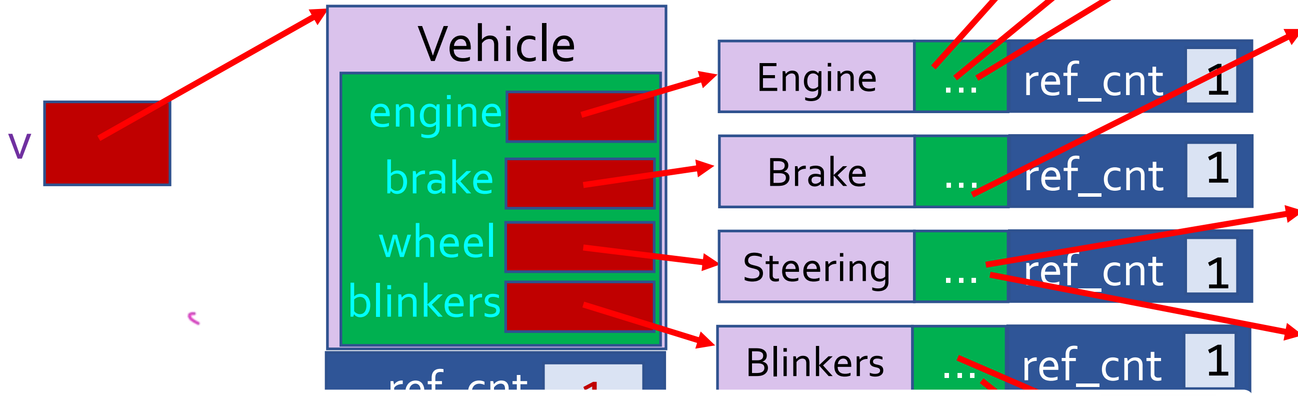
Sarcastic Robot Says:

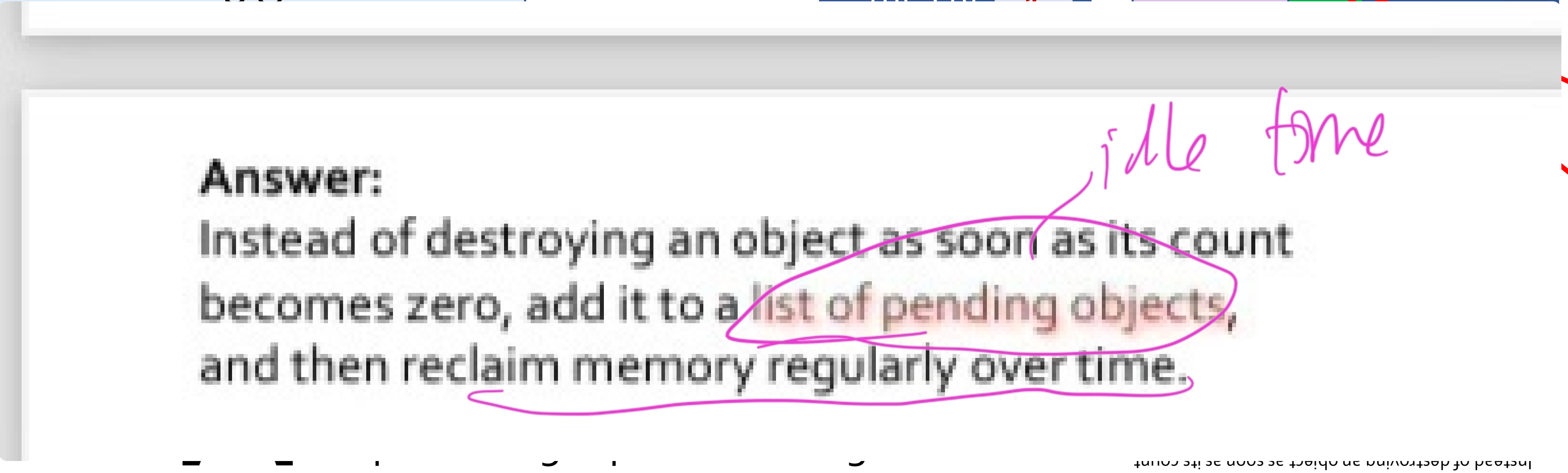
[illegible] that reference counting is also [illegible]

When an object is destroyed (its reference count hits ZERO), all obj
referenced by that object must also have their reference counts

Because of this, removing a single reference can potentially lead to a cascade of objects being freed at once. SLOW!

```
class Vehicle:
  def __init__(self):
    self.engine = Engine()
    self.brake = Brake()
    self.wheel = Steering()
    self.blinke

def game():
→v = Vehicle()
  ...
→v = None
```





idle time

**Answer:**
Instead of destroying an object as soon as its count becomes zero, add it to a list of pending objects, and then reclaim memory regularly over time.

# Garbage Collection: Pick the Winners

We have many objects of diverse sizes with frequent allocations and deletions – what GC scheme(s) are best suited for my situation?

mark and compact to avoid fragmentation

I have lots of objects with cyclical references to each other. What GC scheme(s) should I avoid?

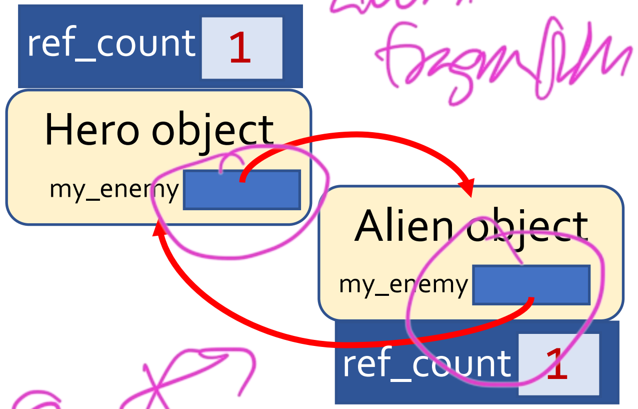
avoid refcount

I am running on a low-RAM device. What GC scheme(s) are best suited for this?

mark and sweep

I am writing a program for a real-time device. What GC scheme(s) are best suited for this?

more lightweight mode ref counts



since complex objs doesn't freed

Q: I have many objects of diverse sizes with frequent allocations and deletions – what GC scheme(s) are best suited for my situation.

A: This situation results in lots of memory fragmentation if you use mark and sweep or reference counting. Mark and compact works better since the objects can be aggregated and memory "holes" can be eliminated in between objects.

Q: I have lots of objects with cyclical references to each other. What GC scheme(s) should I avoid?

A: Avoid reference counting GC, because by definition, two objects that refer to each other will each have reference count of 1, meaning that their reference count will never reach zero even if are no longer referred to by any variables in a program.

Q: I am running on a low-RAM device. What GC scheme(s ) are best suited for this?

Mark and sweep would be best. Mark and compact needs to reserve half the memory for compaction, and reference counting requires extra memory stored with each object to maintain reference counts.

Q: I am writing a program for a real-time device. What GC scheme(s) are best suited for this?

Reference counting would be best (though still not ideal) since it doesn't freeze the computer while GC occurs. Objects are GCed as they are no longer referred to, generally resulting in incremental GC of objects. Use of a queue of pending objects to free to deal with cascades can help even out the load of GC over time.

One more point: mark and sweep (and to a lesser extent, mark and compact) may cause thrashing with OS paging. Why? Here's an answer from former student Victor Chinnappan:

This most likely has to do with locality. It is true that mark-and-sweep is not the only case where thrashing occurs and it doesn't occur in all cases for mark-and-sweep but let's look at an example.

Imagine storing large amounts of data such that RAM is pretty much full. That would lead to the Garbage Collector having to run. Mark-and-sweep will have to traverse through all the memory blocks in our heap. Now if our heap uses a lot of pages and our RAM is not large enough to store all of them, we would have a high page fault rate (paging in and out), hence thrashing.

Note: Fragmentation also makes it worse (less usable blocks and that may be spread apart)

# Garbage Collection Summary

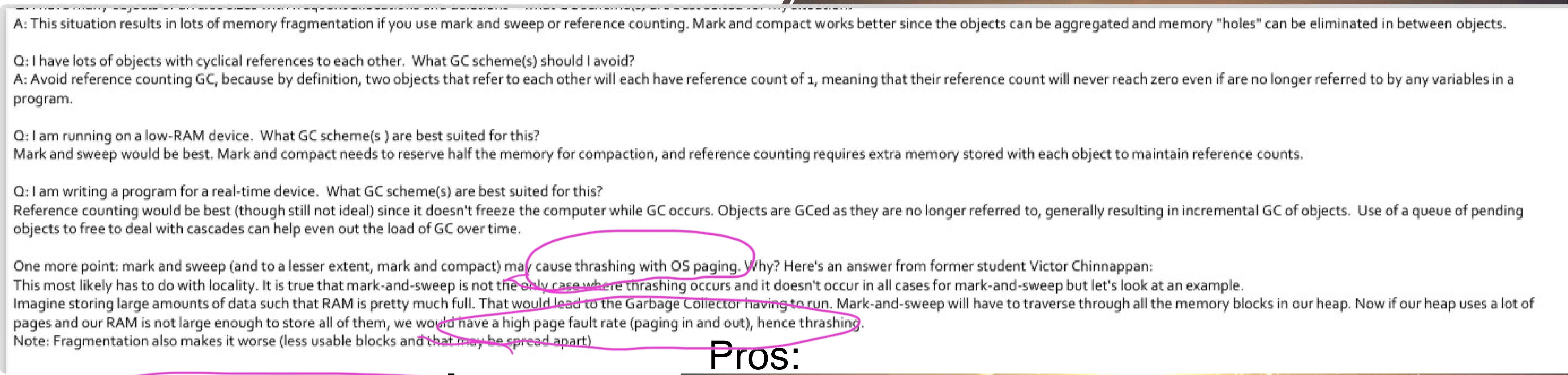
Garbage collection eliminates entire classes of common memory safety bugs.

Obviously, garbage collection adds extra storage and performance overhead, but with clever engineering this can be minimized.

As such, garbage collection is pretty much a de-facto standard in most modern programming languages.

The one area where languages with garbage collection are frowned upon is in real-time devices that need totally predictable execution behavior.

In these environments, languages like C and Rust are used – both of which don't use GC.



Q: I have many objects of all sizes with frequent allocations and deletions – what GC scheme(s) are best suited for my situation.

A: This situation results in lots of memory fragmentation if you use mark and sweep or reference counting. Mark and compact works better since the objects can be aggregated and memory "holes" can be eliminated in between objects.

Q: I have lots of objects with cyclical references to each other. What GC scheme(s) should I avoid?
A: Avoid reference counting GC, because by definition, two objects that refer to each other will each have reference count of 1, meaning that their reference count will never reach zero even if are no longer referred to by any variables in a program.

Q: I am running on a low-RAM device. What GC scheme(s ) are best suited for this?
Mark and sweep would be best. Mark and compact needs to reserve half the memory for compaction, and reference counting requires extra memory stored with each object to maintain reference counts.

Q: I am writing a program for a real-time device. What GC scheme(s) are best suited for this?
Reference counting would be best (though still not ideal) since it doesn't freeze the computer while GC occurs. Objects are GCed as they are no longer referred to, generally resulting in incremental GC of objects. Use of a queue of pending objects to free to deal with cascades can help even out the load of GC over time.

One more point: mark and sweep (and to a lesser extent, mark and compact) may cause thrashing with OS paging. Why? Here's an answer from former student Victor Chinnappan:
This most likely has to do with locality. It is true that mark-and-sweep is not the only case where thrashing occurs and it doesn't occur in all cases for mark-and-sweep but let's look at an example.
Imagine storing large amounts of data such that RAM is pretty much full. That would lead to the Garbage Collector having to run. Mark-and-sweep will have to traverse through all the memory blocks in our heap. Now if our heap uses a lot of pages and our RAM is not large enough to store all of them, we would have a high page fault rate (paging in and out), hence thrashing.
Note: Fragmentation also makes it worse (less usable blocks and that may be spread apart)

# Model

For ref counting

Pros:

simple

usually real-time (since reclamation is usually instant)

more efficient usage since blocks are freed immediately

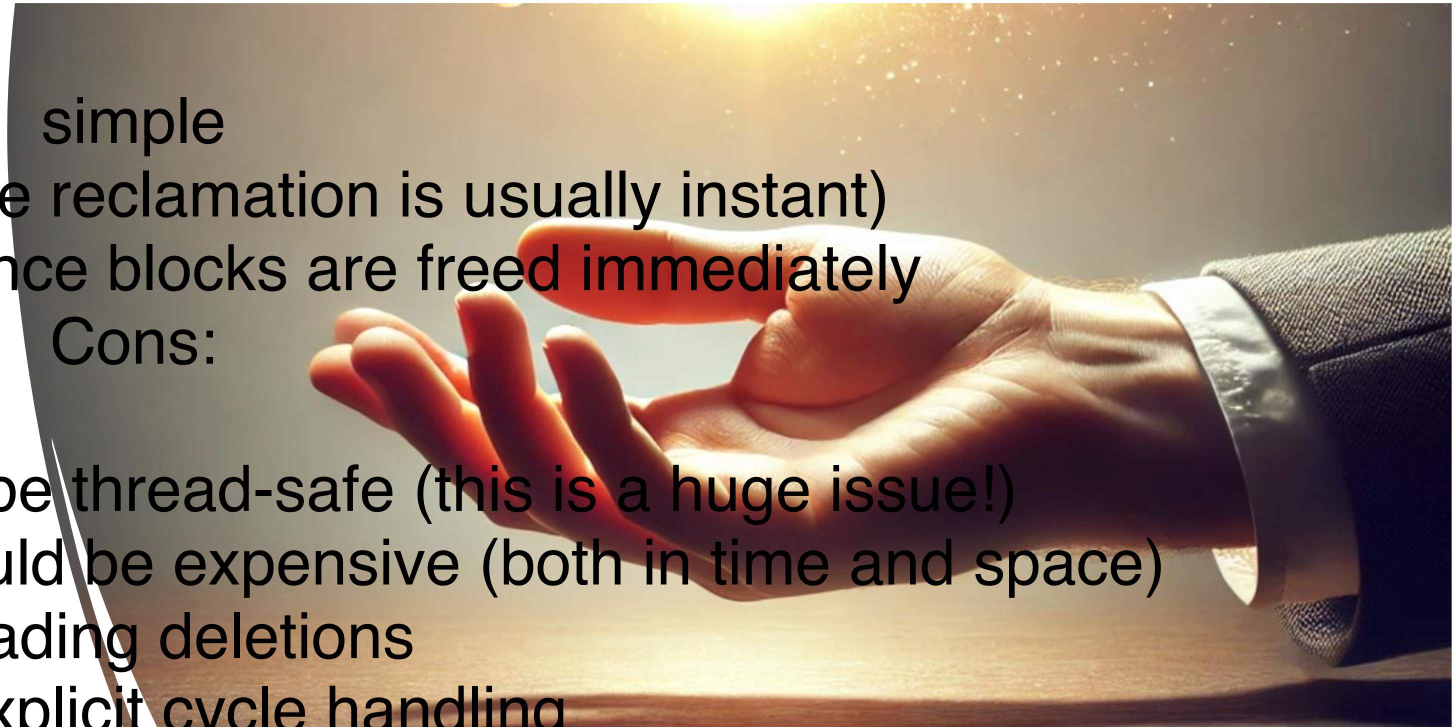
Cons:

updating counts needs to be thread-safe (this is a huge issue!)

updating on every operation could be expensive (both in time and space)

cascading deletions

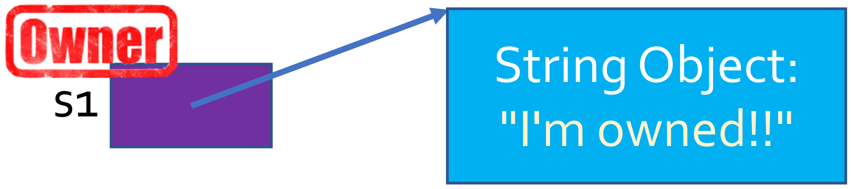
requires explicit cycle handling

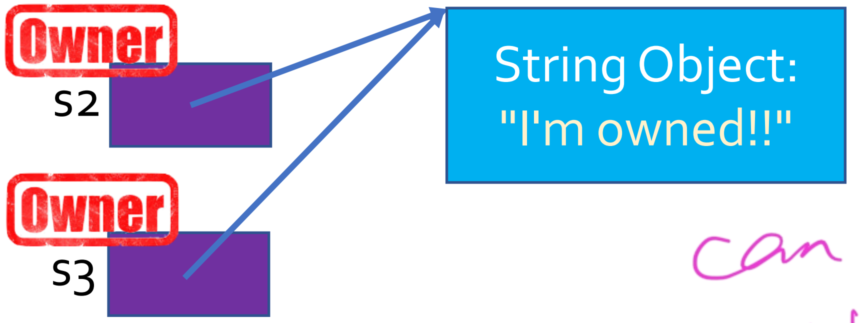


# The Ownership Model

An Alternative to GC -

In the ownership model, every object is "owned" by one or more variables in the program.





Similar to ref counting

C++ smart pointers use ref counting but NOT Rust → only object refs (heap) not stack

When the last owner variable's lifetime ends, the object it owns is freed automatically.

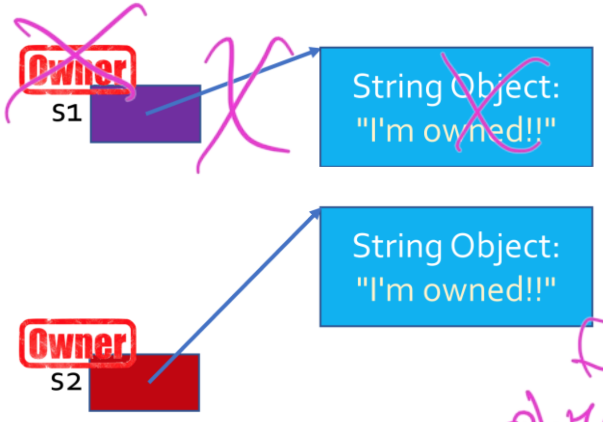
can be determined at compile time ↓ not ref count for each object ?! !

In some implementations, ownership can be transferred (aka "moved") to a new variable, invalidating the old variable!

```
var s1 = new String("I'm owned!")
var s2 = s1
print(s1)  // ERROR!
```

After: var s1 = ...

After: var s2 = s1



# Rust's Ownership Model: Move Semantics

done at compile (ref done at runtime)

zero cost abstraction

In Rust's ownership model, every object is owned by a single variable in the program.

When that variable's lifetime ends, the object it owns is freed.

In Rust, ownership is transferred to a new variable via assignment or parameter passing. After such a transfer, the old owner variable becomes invalid!

object disappears after assignment

```
// Rust example showing ownership concept

fn  foo (s3: String) {
  println!("{}", s3);
}// s3's lifetime ends, string object freed


fn  main() {
  let  s1 = String::from("I'm owned!!");
  let  s2 = s1; // Ownership xferred to s2
  foo(s2);
  println!("{}", s2); // Compiler error!
} // s2's lifetime ends; string object freed
  // Nothing left to free!
```

s3=s2 invalidity

Error

println!("{}",s1)

owner

s1 → "I'm owned"

s2

s3

mem management done at compile time

# Rust's Ownership Model: Move Semantics

Compile time cost ↙ free GC

Rust's ownership model also supports "borrowing" where a variable may refer to an object without taking ownership.

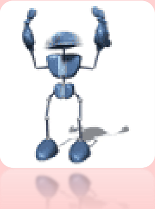
The borrower may request exclusive read/write access (for thread safety) or non-exclusive read-only access.

own
s2 → str
s3 ✗

```
// Rust example showing borrowing

fn  foo (s3: &String) {
  println!("{}", s3);
} // s3 goes out of scope, no object freed!


fn  main() {
  let   s1 = String::from("I'm owned!!");
  let   s2 = s1; // Ownership xferred to s2
  foo(&s2);
  println!("{}", s2); // This is valid!
} // s2 goes out of scope, string object freed
```

← borrow, immutable → read-only

Sarcastic Robot Says:



[illegible] - Rust only uses ownership to track objects, not primitive values.

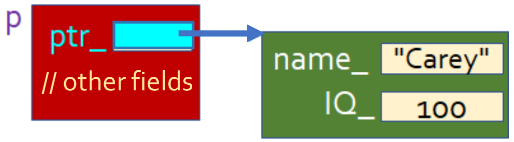
And... you can make copies of values, if [illegible]

cascading frees can still be slow

# C++'s Ownership Model: Smart Pointers

A smart pointer is a C++ class that works like a traditional pointer but also provides automatic memory management.

Each smart pointer object holds a traditional pointer that refers to a dynamically allocated object or array.

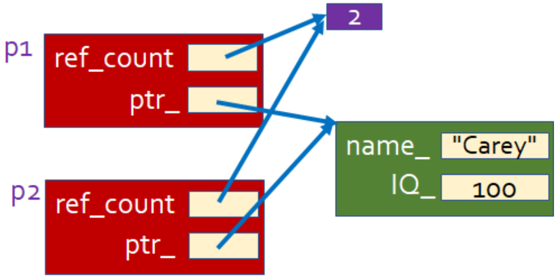


```
class SmartPointer {
 ~SmartPointer()
   { delete ptr_; }
};
```

wrapper

Each smart pointer is an owner of its assigned heap-allocated object, and is responsible for freeing it when it's no longer needed.

When copies are made of a smart pointer, they coordinate and keep track of how many of them refer to the same shared resource.



→ ref count

# std::unique_ptr

A unique_ptr is a smart pointer that exclusively owns the responsibility for freeing a heap-allocated object. When the UP goes out of scope, it frees the object.

```
#include<memory>  // needed for unique_ptr
#include "nerd.h"

int main() {
   std::unique_ptr<Nerd> p = std::make_unique<Nerd>("Carey", 100);

   p->study(); // p acts like a regular ptr!

   std::unique_ptr<Nerd> p2 = p; // ERROR!
} // p goes out of scope → frees the Nerd
```

Nerd * p = new Nerd(

You pass in the parameters for construction of your object to make_unique – it'll automagically forward them to your c'tor!

Instead of using the new command, we call the make_unique function to dynamically allocate RAM and construct a new object.

And when a unique pointer goes out of scope, it auto-deletes the dynamic object it owns.

You can't make copies of a unique_pointer – no duplicating it, or passing it to other functions!

```
// nerd.h
class Nerd {
public:
  Nerd(string name, int IQ)
  void study() { ... }
}
```

auto shared refs for returns

no double refs

move semantics like in Rust

also can't store in data structures, can't pass by value

# std::shared_ptr

A shared_pointer is a smart pointer that shares the responsibility for freeing a heap-allocated object. When the last SP goes away, it frees the object.

```
#include<memory>  // needed for shared_ptr
#include "nerd.h"

std::vector<std::shared_ptr<Nerd>> all_my_nerds;

void keep_track_of_nerd(std::shared_ptr<Nerd> n) {
  all_my_nerds.push_back(n);
} // n goes out of scope

int main() {
  std::shared_ptr<Nerd> p = std::make_shared<Nerd>("Carey", 100);
  keep_track_of_nerd(p);
} // p goes out of scope
// globals like all_my_nerds are destructed
```

vector of shared smart pointers to Nerds

adds to vector

Here's how we define a shared pointer for a Nerd...

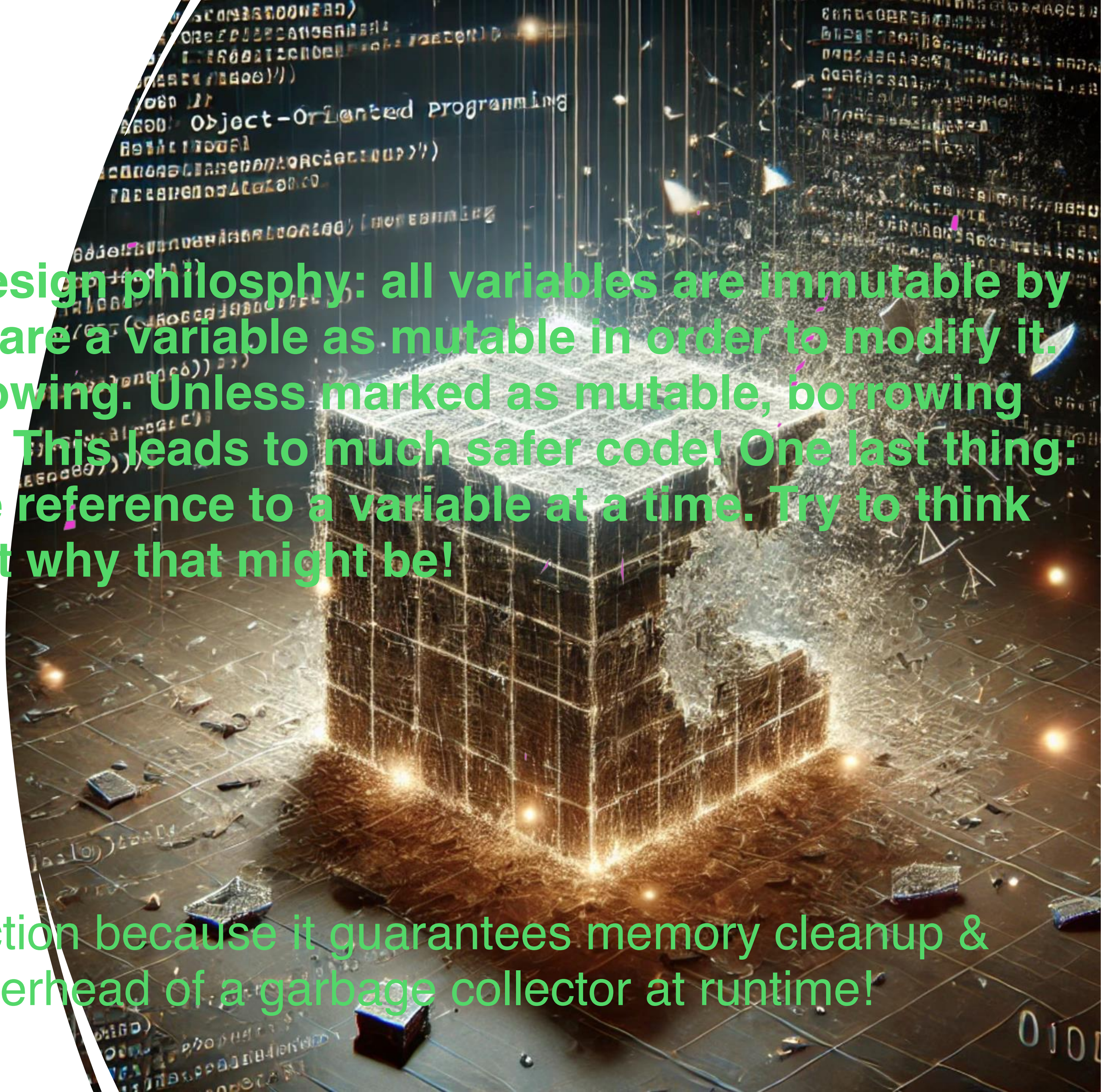
When we pass a shared_ptr by value, it makes another copy of the smart pointer!

make a copy ↓ pass to function

# Memory Safety: What Happens When Objects Die?

**An interesting note on Rust’s design philosphy: all variables are immutable by default! We must explicitly declare a variable as mutable in order to modify it. This idea carries over to borrowing. Unless marked as mutable, borrowing creates an immutable reference. This leads to much safer code! One last thing: we can only have one mutable reference to a variable at a time. Try to think about why that might be!**

Rust :-



This is known as a zero cost abstraction because it guarantees memory cleanup & safety without the additional overhead of a garbage collector at runtime!

# Memory Safety: What Happens When Objects Die?

Many objects hold resources (e.g.: dynamic objects, temp files) which need to be released when their lifetime ends.

There are three ways this is handled in modern languages.

may never be called
gc might never run

## Destructor Methods

Destructors are automatically called when an object's lifetime ends.

It is guaranteed that a destructor will run immediately at this time.

Non-GC languages, e.g.: C++

## Finalizer Methods

An object's finalizer method is called by the garbage collector before it frees the object's memory.

Since garbage collection can occur at any time (or not at all), you can't predict when/if a finalizer will run!

GC languages, e.g.:
C#, Go, Java, Python

## Manual Disposal Method

The programmer adds a "disposal()" method to their class, and updates their code to explicitly call it to force the disposal of resources.

It's like a manually-invoked destructor.

Manual Disposal languages, e.g.:
C#, Java, Swift

# What Happens When Objects Die: Destructors

also have gc but can manually delete objects

Destructors are only used in languages with manual memory management, like C++.

There are deterministic rules that govern when destructors are run, so the programmer can ensure *all* of them will run, and control *when* they run.

Since the programmer can control when they run, you can use destructors to release critical resources at the right times:

e.g., freeing other objects, closing network connections, deleting files, etc.

```
void doSomeProcessing() {
   TempFile *t = new TempFile();

   ...

   if (dont_need_temp_file_anymore())
     delete t;

   ...
}

void otherFunc() {
  NetworkConnection n("www.ucla.edu");

  ...
}
```

ptr

local var

Our object's lifetime is deterministic – the programmer can control exactly when the destructor will run.

Similarly, the destructors for local variables are guaranteed to run when the variables' lifetimes ends.

# What Happens When Objects Die: Finalizers

In GC languages, memory is reclaimed automatically by the garbage collector.

So finalizers are used to release unmanaged resources like file handles or network connections, which aren't garbage collected.

Unlike a destructor, a finalizer may not run at a predictable time or at all, since objects can be garbage collected at any time (or not at all)!

Since they can't be counted on to run, they're considered a last-line of defense for freeing resources, and often not used at all!

We'll learn more about finalizers when we cover Object Oriented Programming.

```
// Java finalization example
public class SomeClass {

  // called by the garbage collector
  protected void finalize() throws Throwable
  {
    // Free unmanaged resources held by SomeObj
    ...
  }
}
```

not to free memory, but to free non RAM resources

```
# Python finalizer method
class SomeClass :
  ...

  # called by the garbage collector
  def __del__(self):
    # Finalization code goes here
    ...
```

gc frees memory and then calls finalizers

# What Happens When Objects Die: Disposal Methods

A disposal method is a function that the programmer must manually call to free non-memory resources (e.g., network connections)

You use disposal methods in GC languages because you can't count on a finalizer to run!

Disposal provides a guaranteed way to release unmanaged resources when needed.

But... If the programmer forgets to call Dispose(), it'll never run!

```
// C# dispose example
public class FontLoader : IDisposable
{
  ...

  public void Dispose()
  {
     // do manual disposal here, e.g., free
     // temp files, close network sockets, etc.
  }
}

...

var f = new FontLoader(...);
... // use f to draw fonts
f.Dispose();
```

interface

manually
call finalizers

# A Final Word on C and C++ and Safety

As much as I like C++, it's by far the most memory unsafe language in wide use today!

C++:

- Allows out-of-bound array indexes and unconstrained pointer arithmetic
- Allows casting variables to incompatible types
- Allows use of uninitialized variables/pointers
- Is susceptible to memory leaks
- Allows use of dangling pointers to dead objects

# Classify That Language: Memory Safety

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
}

class Apartment {
    let unit: String
    init(unit: String) { self.unit = unit }
    var tenant: Person?
}

var pers:Person? = Person(name: "Dean Boelter")
var apt:Apartment? = Apartment(unit: "11C")
pers!.apartment = apt
apt!.tenant = pers

pers = nil
apt = nil
```

cyclical owner

For some reason, the pers and apt objects never get finalized in this program.

What type of GC might this language be using:

Mark and Sweep

Mark and Compact

[✓ Reference Counting]

# Classify That Language: Memory Safety

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
}

class Apartment {
    let unit: String
    init(unit: String) { self.unit = unit }
    var tenant: Person?
}

var pers:Person? = Person(name: "Dean Boelter")
var apt:Apartment? = Apartment(unit: "11C")
pers!.apartment = apt
apt!.tenant = pers

pers = nil
apt = nil
```

This creates a cycle between the objects where they both point at each other.

A Mark and Sweep collector would have no problem GCing these objects here… So the language must be using Reference Counting.

or mark and compact

For some reason, the pers and apt objects never get finalized in this program.

What type of GC might this language be using:

Mark and Sweep

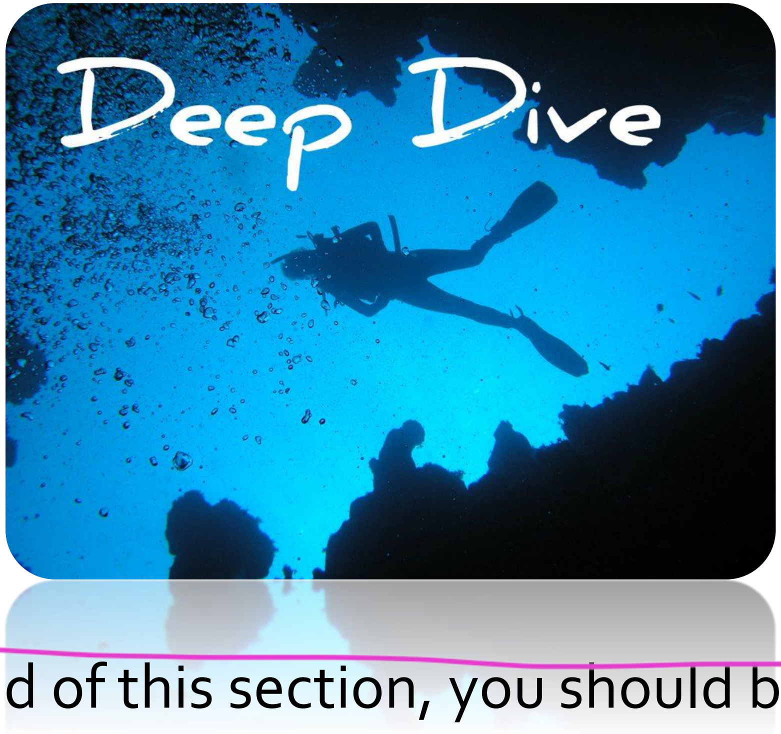
Mark and Compact

Reference Counting

This is Swift!

need other methods to GC this

# Mutability/Immutability



By the end of this section, you should be able to:

Take a new language and understand what features it has to create constant variables and values.

Understand how those features can let you write safer code.

# Mutability/Immutability

## What’s the big picture?

Immutability is the property that a variable/value/object is read-only, and it can't be changed (aka "mutated") once initialized.

Rather than modifying an existing value, when a new value is needed, you construct a new object with changes, based on the original.

Immutability is provided by language features, not by hardware-level protection!

Immutability has many benefits, including eliminating may bugs, speeding garbage collection, etc!

# Immutability – Four Approaches

### Class Immutability

The programmer can designate that all objects of a class are immutable after construction.

```
def main():  # Python
  s = "Hello!"
  s[0] = 'J'  # ERROR!
```

### Object Immutability

The programmer can designate some objects of a particular class as immutable – mutations are blocked to those objects!

```
int main() {
  Nerd j("Joe",200); // mutable!
  const Nerd n("Carey",100);
  n.setIQ(120); // ERROR!
}
```

### Assignability Immutability

The programmer can designate that a variable may not be re-assigned to a new value - but mutations *can* be made to the original referred-to object!

```
public static void someFunc() {
  final Nerd n = Nerd("Carey",100);
  n = Nerd("Joe,200); // ERROR!
  n.setIQ(120);        // OK!!!
}
```

### Reference Immutability

The programmer can prevent a mutable object from being mutated via a reference that's marked as immutable

```
void examine(const Nerd& n);

int main() {
  Nerd j("Joe",200);
  examine(j);
```

CHALLENGE! Which of these approaches can be implemented with C++'s const keyword?

Answer: Object immutability, assignability mutability, and reference immutability

# Why Immutability?

## Fewer Bugs

### Eliminates Aliasing Bugs

f(x,x);
If f() can't modify x,
then no aliasing bugs!

### Reduces multithreading bugs

If a value can't change,
you can't have race
conditions!

### Eliminates Identity Variability Bugs

```
map[x] = y;
x.change_identity();
cout << map[x]; // ???
```

### Eliminates Temporal Coupling Bugs

Circle c = new Circle();
~~c.setRadius(10);~~
c.getArea();  // ??

Temporal Coupling Bug:

A bug where the programmer does some initialization out of order – or not at all -  resulting in use of an incomplete object.

## Improved Code Quality

### Absence of Hidden Side Effects

Makes programs
easier to read and
reason about

### Makes Testing Easier

There are far fewer
failure modes since
objects are frozen

### Enables Runtime Optimizations

The compiler can make
assumptions about
objects that can't change

### Enables Easy Caching

Objects can be cached
without concern their
values have changed

### Ensures Atomicity of Failure

Objects are never left in an
inconsistent state by definition

# Classify That Language: Immutability

```
struct Point {
  x: isize,
  y: isize,
}

impl Point {
  fn new(x: isize, y: isize) -> Self {
    Self { x, y }
  }
  fn change(&mut self, x: isize, y: isize)
   { self.x = x; self.y = y; }
}

fn main() {
  let p = Point::new(0, 0);

  p.change(10,20);

  p = Point::new(1, 2);
}
```

cannot borrow `p` as mutable

cannot assign twice to immutable variable `p`

The following program generates two compiler errors.

What immutability approach(es) are used by the following language?

# Classify That Language: Immutability

```
struct Point {
  x: isize,
  y: isize,
}

impl Point {
  fn new(x: isize, y: isize) -> Self {
    Self { x, y }
  }
  fn change(&mut self, x: isize, y: isize)
   { self.x = x; self.y = y; }
}

fn main() {
  let p = Point::new(0, 0);

  p.change(10,20);

  p = Point::new(1, 2);
}
```

In this language, let indicates that immutability is to be applied.

cannot borrow `p` as mutable

This error indicates that the language provides Object Immutability.

cannot assign twice to immutable variable `p`

This error indicates that the language provides Assignment Immutability.

The following program∧ generates two compiler errors.

What immutability approach(es) are used by the following language?

This is Rust!

# Mutability

You might remember the concept of immutability from our discussion of functional programming: it’s used to describe objects that are “read only.” In other words, once an immutable object has been defined, it cannot be changed.

Instead, we simply construct a new object based on the original, including any changes we would like to make. There are tons of benefits to immutability including eliminating bugs, speeding up garbage collection, and more! Let’s take a closer look.

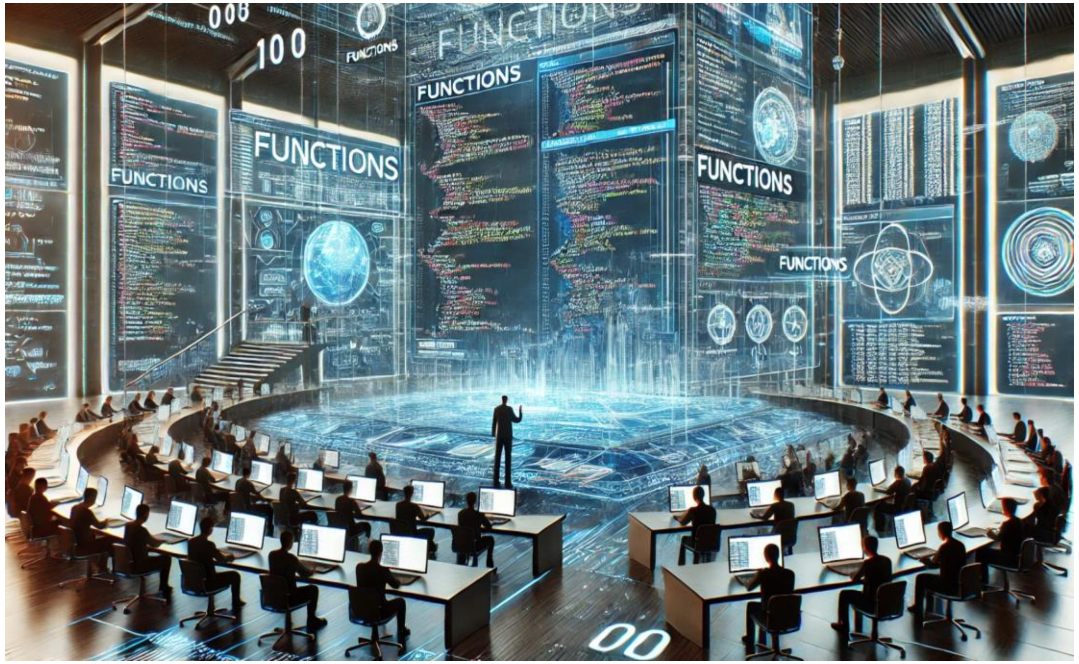
There are four approaches to immutability:

Class immutability: The programmer can designate that all objects of a class are immutable after construction.
Object immutability: The programmer can designate some objects of a particular class as immutable –mutations are blocked to those objects!
Assignability immutability: The programmer can designate that a variable may not be re-assigned to a new value - but mutations can be made to the original referred-to object!
Reference immutability: The programmer can prevent a mutable object from being mutated via a reference that’s marked as immutable
There are tons of benefits!

eliminates aliasing bugs
reduces race conditions in multithreaded code
eliminates identity variability bugs
elminates temporal coupling bugs
removes side effects, making programs easier to reason about
makes testing easier
enables runtime optimizations
enables easy caching
objects are never left in an inconsistent state by definition

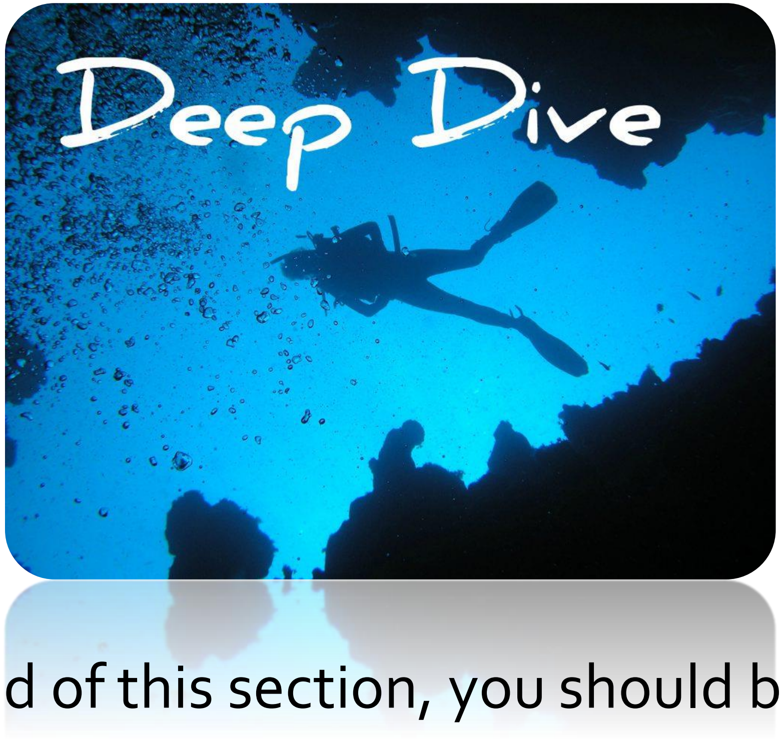
# Data-Function-palooza

+



This section covers Variable Binding Semantics and Parameter Passing - two intimately-related topics that bridge both our data and function units.

# Variable Binding and Parameter Passing Semantics



By the end of this section, you should be able to:

Take a new language and understand how it associates variable names with values and passes parameters to functions!

Understand the implications of each approach to avoid common bugs.

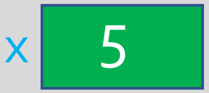
# Binding and Parameter Passing Semantics

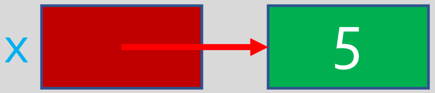
## What’s the big picture?

Binding Semantics is the term we use to describe the different ways that languages associate variable names (e.g., x) with the actual storage in RAM that holds their values (e.g., 5).

```
// C++
int main() {
  int x = 5;
}
```

```
# python
def main():
  x = 5
```





For instance, some languages directly associate a variable name with its value.

Other languages associate a variable name with a pointer to a value stored elsewhere.

Each approach has implications for how you write code, pass variables to functions, and what bugs you run into!

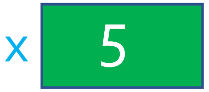
# Variable Binding Semantics

Binding Semantics describe how a variable name is bound to a storage+value.

## Value Semantics

A variable name is directly bound to the storage that holds the value

```
int main() {
  int x = 5;

}
```

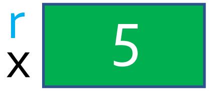


C++, Go, Java

## Reference Semantics

A variable name is directly bound to another variable's storage, like an alias

```
int main() {
  int x = 5;
  int &r = x;
}
```

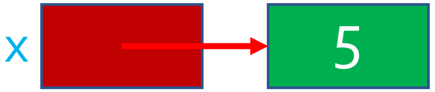


C++, C#, PHP, Rust

## Object Reference Semantics

A variable name is bound to a pointer that points to an object/value

```
def main():
  x = 5
```



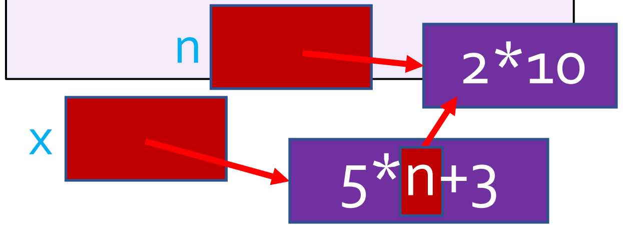
Java, JavaScript, Python, Ruby

(And C++ via pointers)

## Name/Need Semantics

A variable name is bound to a pointer that points to an expression graph that can be evaluated to get a value

```
main = do
  let n = 2*10
  let x = 5*n+3
```



Haskell, R, Scala

# Parameter Passing Semantics

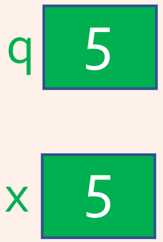
Parameter Passing Semantics are directly related to Binding Semantics!

## Value Semantics

Pass by Value
(aka Pass by Copy)

The formal parameter gets a distinct copy of the argument's value/object

```
int f(int q) {…}
int main() {
  int x = 5;
  f(x);
}
```

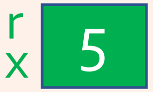


## Reference Semantics

Pass by Reference

The formal parameter is bound to the argument's storage, like an alias

```
int f(int &r) {…}
int main() {
  int x = 5;
  f(x);
}
```

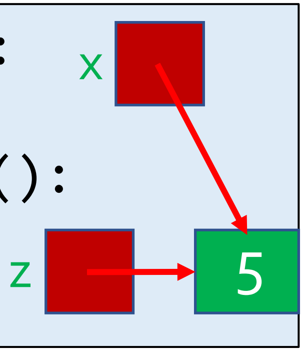


## Object Reference Semantics

Pass by Object Reference

The formal parameter is a pointer that points to the argument object

```
def f(x):
  ...
def main():
  z = 5
  f(z)
```

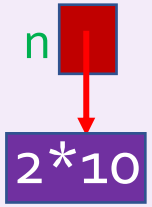


## Name/Need Semantics

Pass by Name
Pass by Need

The formal parameter is a pointer that points to an expression graph

```
f n = 5*n+3

main = do
  let z = f (2*10)
```



# Variable Binding Semantics

Let's learn the following about each approach using the following framework:

How does "initial binding" of the variable work

```
int main() {
  Dog d = Dog("Koda");
  Dog e = Dog("Fido");
  ...

}
```

What happens when we do a "variable update"

```
int main() {
  Dog d = Dog("Koda");
  Dog e = Dog("Fido");
  d = e;
  ...
}
```

What happens when we do a "variable mutation"

```
int main() {
  Dog d = Dog("Koda");
  Dog e = d;

  d.set_bark(10);
}
```

# Value Semantics

Each variable name is directly "bound" to storage
on the stack that holds the variable's value.

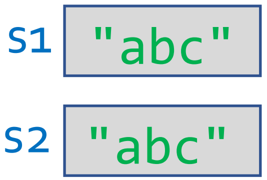
x 5

How does "initial binding" of the variable work

What happens when we do a "variable update"

What happens when we do a "variable mutation"

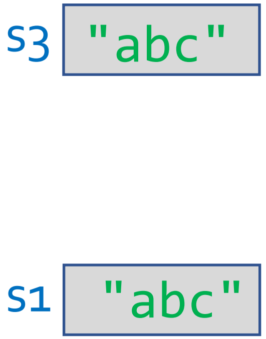
```
int main() {
  string s1 = "abc";
  string s2 = s1;
  ...
}
```



s1 "abc"

s2 "abc"

```
void foo(string s3) {
  ...
}

int main() {
  string s1 = "abc";
  foo(s1);
}
```



s3 "abc"

s1 "abc"

# Value Semantics

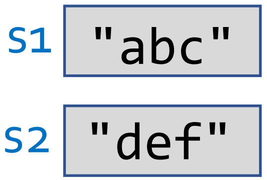
Each variable name is directly "bound" to storage
on the stack that holds the variable's value.

How does "initial binding" of the variable work

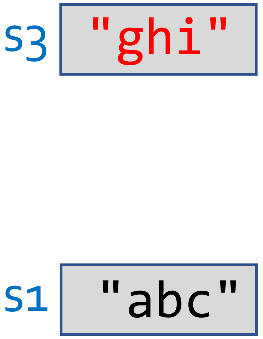
What happens when we do a "variable update"

What happens when we do a "variable mutation"

```
int main() {
  string s1 = "abc";
  string s2 = s1;
→s2 = "def";
}
```



```
void foo(string s3) {
→s3 = "ghi"
}

int main() {
  string s1 = "abc";
  foo(s1);
}
```



# Value Semantics

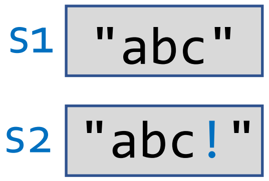
Each variable name is directly "bound" to storage on the stack that holds the variable's value.

How does "initial binding" of the variable work

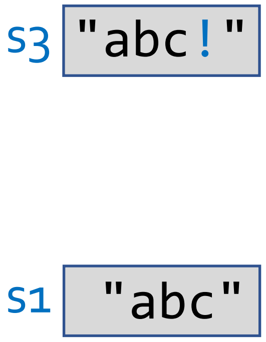
What happens when we do a "variable update"

What happens when we do a "variable mutation"

```
int main() {
  string s1 = "abc";
  string s2 = s1;
  s2.append("!");
}
```



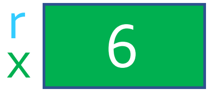
```
void foo(string s3) {
  s3.append("!");
}

int main() {
  string s1 = "abc";
  foo(s1);
}
```



Takeaway: With Value Semantics, each variable has its own separate storage, so assignment/mutation of one variable doesn't affect the others.

# Reference Semantics

A reference variable acts as an alias for an existing variable, allowing you to access/modify the original variable's value through that alias.

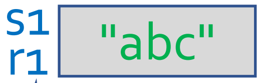


How does "initial binding" of the variable work

What happens when we do a "variable update"

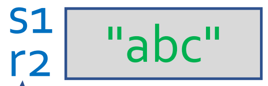
What happens when we do a "variable mutation"

```
int main() {
  string s1 = "abc";
  string &r1 = s1;
  ...
}
```



The reference is an alias for the original variable!

```
void foo(string &r2) {
  ...
}

int main() {
  string s1 = "abc";
  foo(s1);
}
```



The reference is an alias for the original variable!

# Reference Semantics

A reference variable acts as an alias for an existing variable, allowing you to access/modify the original variable's value through that alias.

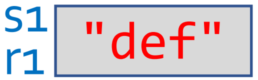
How does "initial binding" of the variable work

What happens when we do a "variable update"

What happens when we do a "variable mutation"

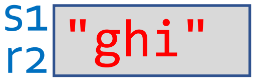
```
int main() {
  string s1 = "abc";
  string &r1 = s1;
→r1 = "def";
}
```

Notice that changes to r1 actually change s1.



```
void foo(string &r2) {
→r2 = "ghi";
}

int main() {
  string s1 = "abc";
  foo(s1);
}
```

Notice that changes to r2 actually change s1.



And the change persists even after we return from the foo() function!

# Reference Semantics

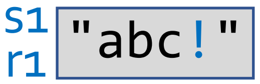
A reference variable acts as an alias for an existing variable, allowing you to access/modify the original variable's value through that alias.

How does "initial binding" of the variable work

What happens when we do a "variable update"

What happens when we do a "variable mutation"

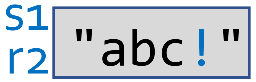
```
int main() {
  string s1 = "abc";
  string &r1 = s1;
→ r1.append("!");
}
```



Notice that changes to r1 actually change s1.

```
void foo(string &r2) {
→ r2.append("!");
}

int main() {
  string s1 = "abc";
  foo(s1);
}
```

Notice that changes to r2 actually change s1.



And the change persists even after we return from the foo() function!

Takeaway: With reference semantics, both assignment (e.g., r1 = "def") and mutation (e.g., r2.append("!")) change the referred-to variable (e.g., s1).

# Reference Semantics: Examples

Let's see how references work in Swift and C#:

```
// References in Swift
func foo (s: inout String) {
  s.append("!")
}

var message = "abc"
foo(s: &message)
print(message) // Output: abc!
```

And we use the inout keyword for the formal parameter.

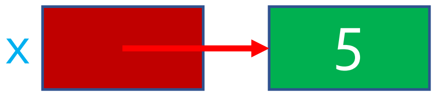
In Swift we use an & to indicate a variable is passed by reference.

```
// References in C#
class Program
{
  static void foo(ref string s) {
    s += "!";
  }

  static void Main() {
    string message = "abc";
      foo(ref message);
      Console.WriteLine(message); // Output: abc!
  }
}
```

In C# we use ref in both places.

# Object Reference Semantics

Each variable name is bound to a pointer that points to a separate object/value.



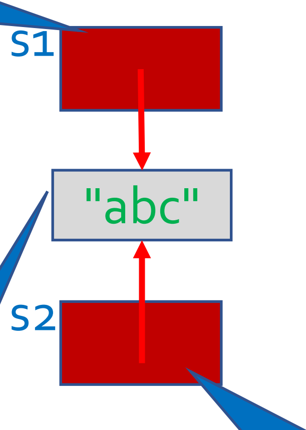
How does "initial binding" of the variable work

What happens when we do a "variable update"

What happens when we do a "variable mutation"

```
def main
   s1 = "abc"
   s2 = s1
   ...
end
```

The object reference variable is a pointer.



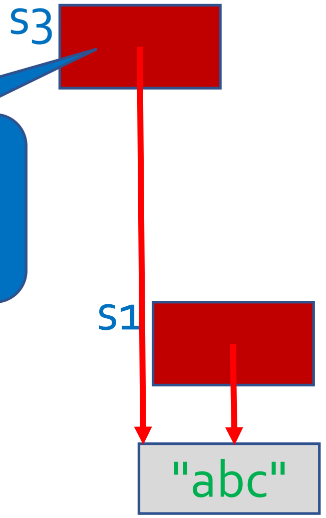
#1: When we define a new object reference (s2) and assign it to an existing one (s1)...

#2: The new object reference copies the address in the old pointer...

#3: So they both point at the same value/object in memory.

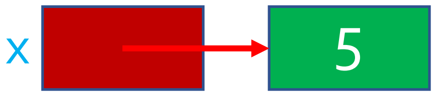
```
def foo(s3)
  ...
end

def main
   s1 = "abc"
   foo(s1)
```

The new object reference points at our original object/value.



# Object Reference Semantics

Each variable name is bound to a pointer that points to a separate object/value.



This assignment points our s3 pointer at a new value!

How does "initial binding" of the variable work

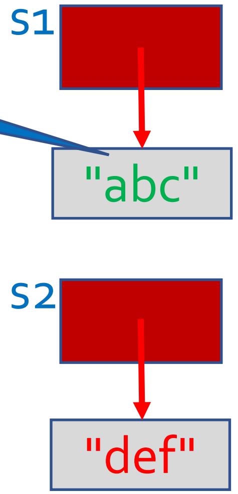
What happens when we do a "variable update"

What happens when we do a "variable mutation"

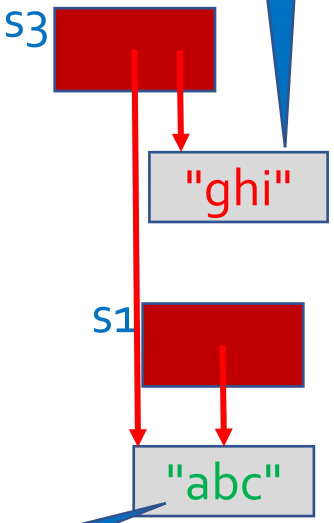
```
def main
   s1 = "abc
   s2 = s1
→ s2 = "def"
end
```

It has no effect on s1, which still points to "abc"!

This variable update points our s2 pointer at a new value!



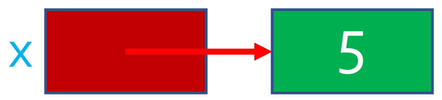
```
def foo(s3)
→s3 = "ghi"
end

def main
   s1 = "abc"
   foo(s1)
end
```



It has no effect on s1, which still points to "abc"!

# Object Reference Semantics

Each variable name is bound to a pointer that points to a separate object/value.

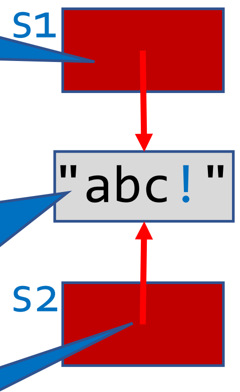


How does "initial binding" of the variable work

What happens when we do a "variable update"

What ha en we do a "va tion"

```
def main
  s1 = "abc"
  s2 = s1
  s2.concat("!")
end
```

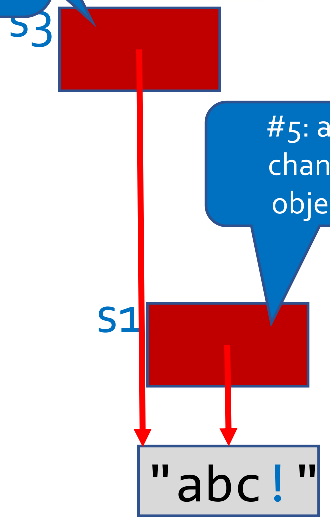


#1: Notice that mutating calls to s2's object…

#2: actually change s1's object too.

#3: Because they both refer to the same string object!

```
def foo(s3)
  s3.concat("!")
end

def main
  s1 = "abc"
  foo(s1)
end
```



#4: Notice that mutating calls to s3's object…

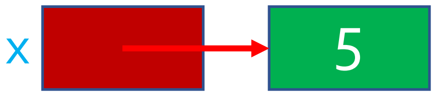
#5: actually change s1's object too!

#6: Because they both refer to the same string object!

And the change persists even after we return from the foo() function!

# Object Reference Semantics

Each variable name is bound to a pointer that points to a separate object/value.

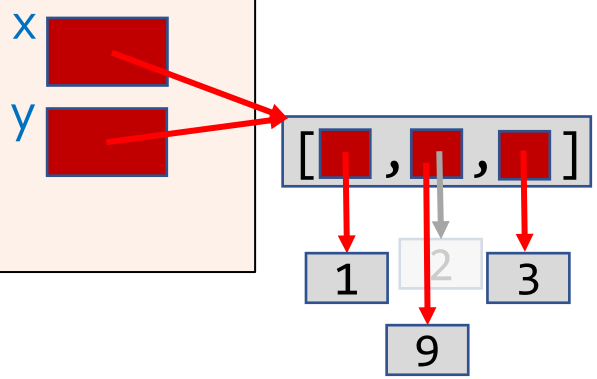


How does "initial binding" of the variable work

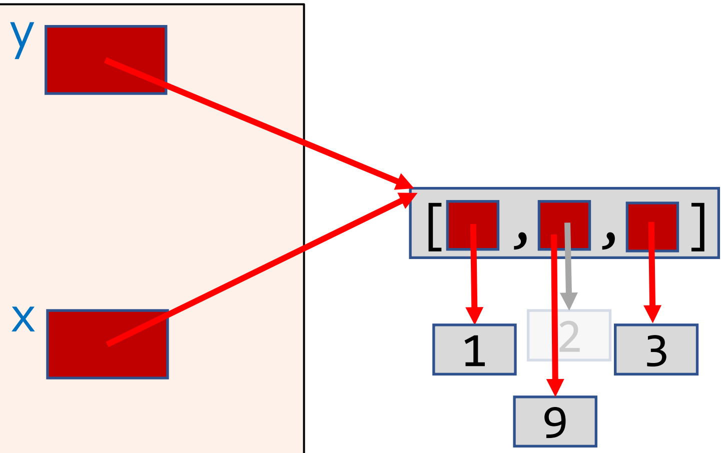
What happens when we do a "variable update"

What happens when we do a "variable mutation"

```
def main
  x = [1,2,3]
  y = x
  x[1] = 9
end
```



```
def foo(y)
  y[1] = 9
end

def main
  x = [1,2,3]
  foo(x)
end
```



Takeaway: When two object references point to the same object, assignment of one to a new value does not change the other, but mutation impacts both.

# Object Reference Challenge!

Consider these programs in Python and Ruby, and their output:

```
# Python
def main():
  x = [1, 2]
  y = x
→ x += [3]
  print(x)
  print(y)
```

```
# Ruby
def main
  x = [1, 2]
  y = x
→ x += [3]
  puts x
  puts y
end
```

```
[1, 2, 3]
[1, 2, 3]
```

```
[1, 2, 3]
[1, 2]
```

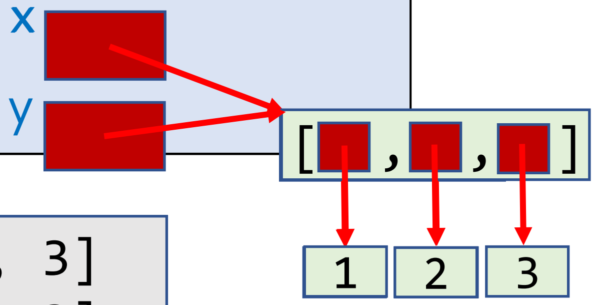
Why does += change the shared list of x and y in Python, but not in Ruby?

# Object Reference Challenge!

Consider th thon and Ruby, and

This is a variable mutation!

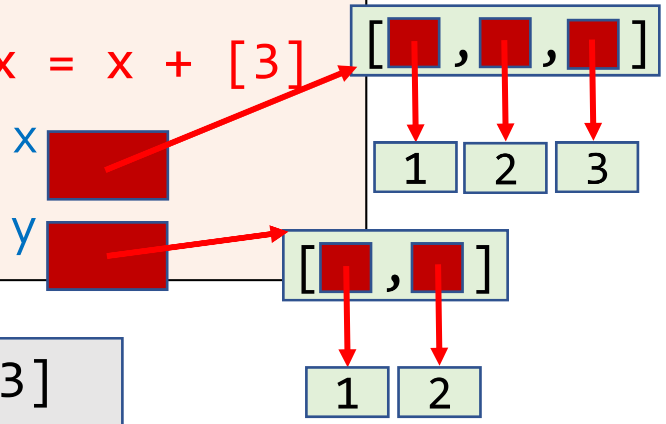
```
# Python
def main():
  x = [1,   ]
  y = x
  x += [3]    # x.append(3)
  print(x)
  print(y)
```



```
[1, 2, 3]
[1, 2, 3]
```

This is a variable update!

```
# Ruby
def main
  x = [1, 2]
  y = x
  x += [3]    # x = x + [3]
  puts x
  puts y
end
```



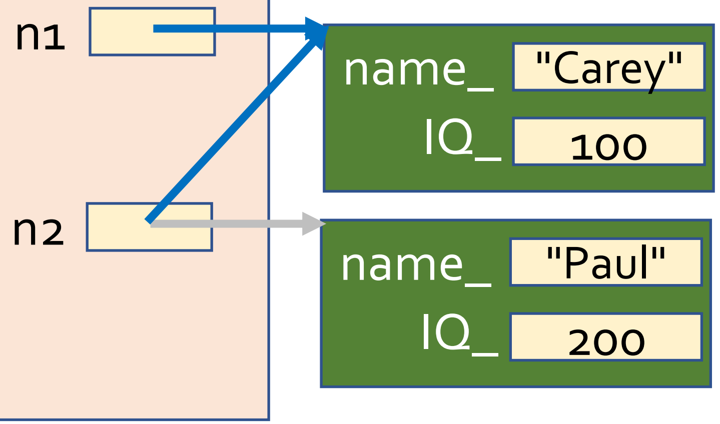
```
[1, 2, 3]
[1, 2]
```

Why does += change the shared list of x and y in Python, but not in Ruby?

Answer: Because in Python, x += y is syntactic sugar for the mutating call x.append(y), but in Ruby, x += y is syntactic sugar for the variable update x = x + y!

# Object References: Java

```
public class Nerd {
  Nerd(String name, int iq) {
    name_ = name;
    iq_ = iq;
  }
  …
  private String name_;
  private int iq_;
}

public class SomeOtherClass {
 void someFunc() {
   Nerd n1 = new Nerd("Carey",100);
   Nerd n2 = new Nerd("Paul",200);
   n2 = n1;
   …
 }
}
```



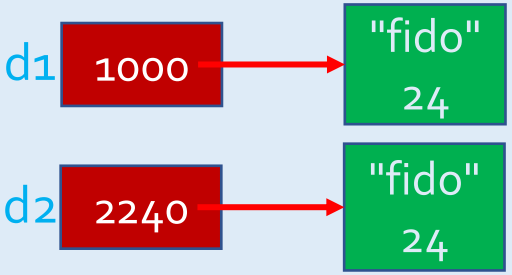
Java uses object reference semantics for all objects… but not for primitive types like ints and doubles.

And in fact, object reference semantics is the dominant paradigm in most modern languages:

C#, Java, Javascript, Python, etc.

# Object Reference Semantics: Testing for Equality

```
# Python object identity vs. equality
class Dog:
 def __init__(self, name, weight):
  self.name = name
  self.weight = weight

 def __eq__(self,other):
  return self.name == other.name and \
         self.weight == other.weight

def main():
 d1 = Dog("Fido",24)
 d2 = Dog("Fido",24)

 if d1 == d2:
  print("d1 has object equality with d2")
 if d1 is d2:
  print("d1 and d2 have the same identity")
 if d1 is d1:
  print("d1 and d1 have the same identity")
```

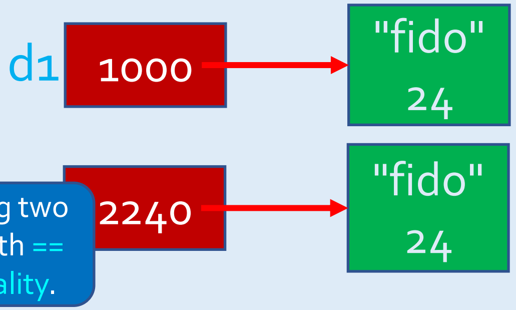


CHALLENGE! When we compare two object references with == what happens?

# Object Reference Semantics: Testing for Equality

```
# Python object identity vs. equality
class Dog:
 def __init__(self, name, weight):
  self.name = name
  self.weight = weight

 def __eq__(self,other):
  return self.name == other.name and \
         self.weight == other.weight

def main():
 d1 = Dog("Fido",24)
 d2 = Dog("Fido",24)

 if d1 == d2:
  print("d1 has object equality with d2")
 if d1 is d2:
  print("d1 and d2 have the same identity")
 if d1 is d1:
  print("d1 and d1 have the same identity")
```

Dunder (aka "double underscore") functions like __eq__ enable Python objects to customize how they're compared, printed, iterated over, etc.



In Python, comparing two object references with == tests for object equality.

In python, comparing two object references with "is" tests for the same object identity.

You might also see folks using ===, which is the same as "is".

CHALLENGE! When we compare two object references with == what happens?

There are two concepts of equality when it comes to object references:

Object Identity: Do two object references refer to the same object at the same address in RAM.

Object Equality: Do two object references refer to objects that have equivalent values (even if they're different objects in RAM).

d1 has object equality with d2
d1 and d1 have the same identity

# : Testing for Equality

```
public class Dog {
  ...
  public Boolean equals(Dog other) {
    return name_.equals(other.name_) &&
           weight_ == other.weight_;
  }

  String name_;
  int weight_;
}

public OtherClass {
 public static void main(String args[]) {
  Dog d1 = new Dog("Fido",24);
  Dog d2 = new Dog("Fido",24);

  if (d1.equals(d2))
   System.out.println("d1 & d2 have equality");
  if (d2 == d1)
   System.out.println("d1 & d2 have same identity");
  if (d1 == d1)
   System.out.println("d1 & d1 have same identity");
 }
}
```

Ok, here's the Java version!

In Java, we use the equals() method to test if two objects are logically equal.

In Java, comparing two object references with == tests for object identity.

Object Identity: Do two object references refer to the same object at the same address in RAM.

Object Equality: Do two object references refer to objects that have equivalent values (even if they're different objects in RAM).

```
d1 & d2 have equality
d1 and d1 have same identity
```

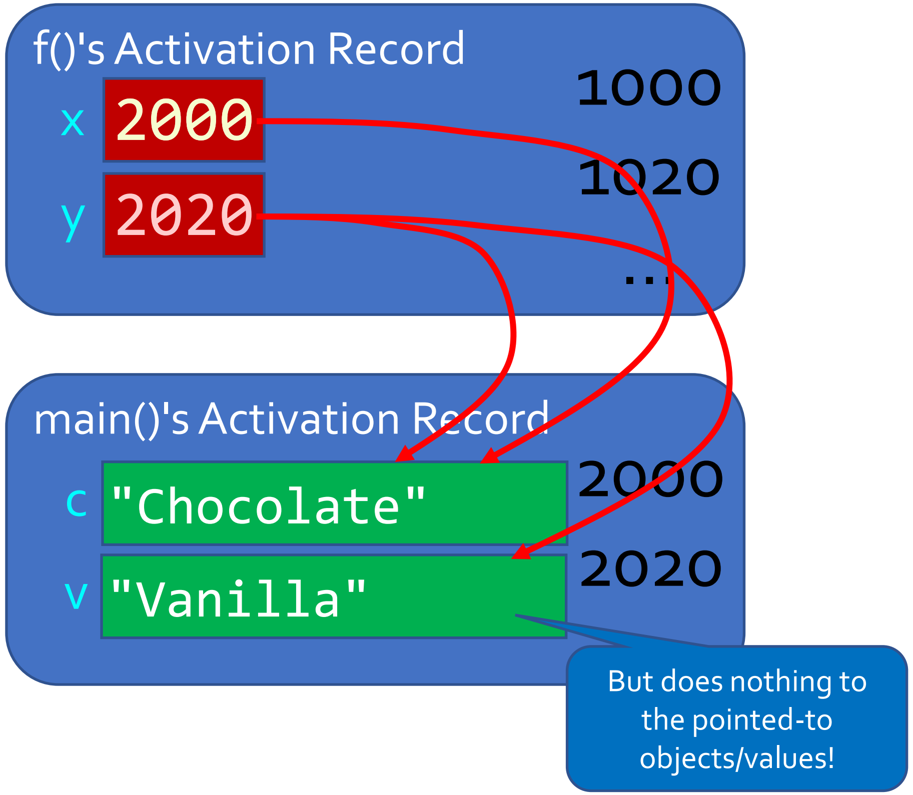
# Pointers: A Type of Object Reference

When we pass a pointer to a function, it's identical to passing by object reference! Let's see!

```
void f(string *x, string *y) {
  y = x;
}

int main() {
  string c = "Chocolate";
  string v = "Vanilla";
   2000 2020
  f(&c, &v);
  cout << v; // Vanilla
}
```

Just like assignments with object references, this just copies the pointer from x into y.

When we use & to get the address of a value/object, it gives us a pointer – that's basically an object reference!



But does nothing to the pointed-to objects/values!

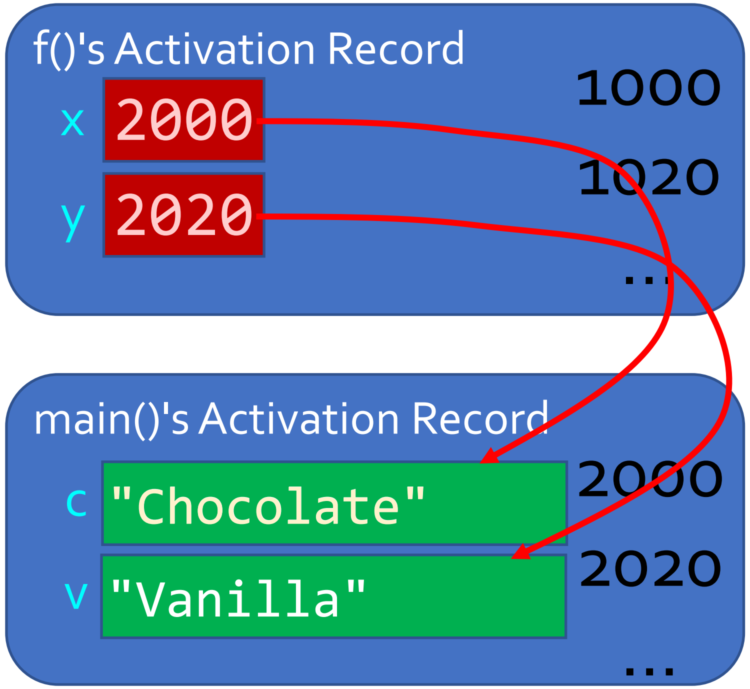
# Pass by Pointer: A Type of Pass by Object Reference

Use of the * lets us read/write the pointed-to objects!

Ok, but what if we use *s to dereference our pointers?

Then we can read/write the pointed-to object itself!

```
void f(string *x, string *y) {
  *y = *x;
}

int main() {
  string c = "Chocolate";
  string v = "Vanilla";
   2000 2020
  f(&c, &v);
  cout << v; // Chocolate
}
```



Moral: Using dereferenced pointers work the same as reference semantics in C++!

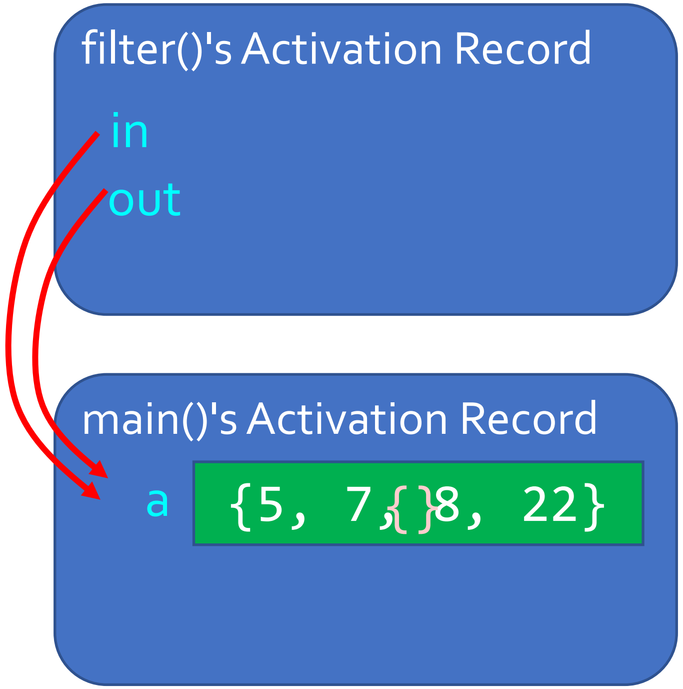
# Aliasing

"Aliasing" occurs when two parameters to a function unknowingly refer to the same value/object and the function modifies it.

Our intent here is to clear variable out, but out and in refer to the same variable a! So this clears our input a before it can be processed!

```
void filter(set<int> &in
            set<int> &out) {
  out.clear();
  for (auto x: in)
    if (is_prime(x)) out.insert(x);
}

int main() {
  set<int> a;
  ... // fill up a with #s
  filter(a, a); // wrong result!
}
```

Notice we're passing in a for both parameters!



Aliasing can occur any time you use references or object references.

It can cause subtle and difficult to find bugs – let's see!

To avoid aliasing, prefer returning new objects instead of mutating passed-in objects.

# Name Semantics

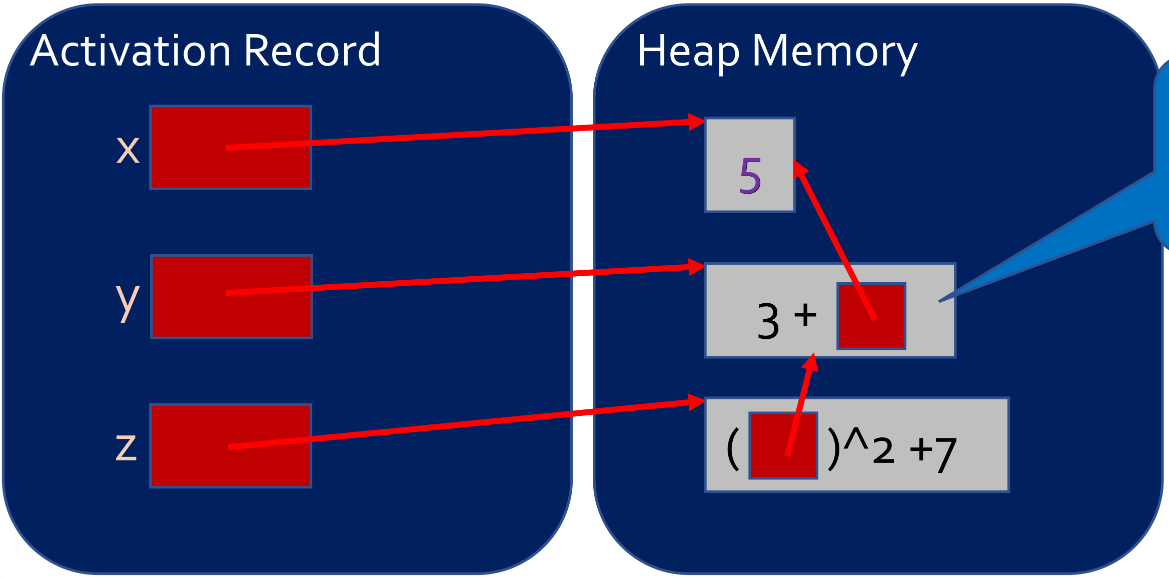
Languages with name semantics bind each variable name to the equivalent of an expression graph, which once evaluated, yields the final value of the variable.

When a variable's value is needed (e.g., to be printed), the expression represented by the graph is "lazily evaluated" and a value is produced.

Any computation in the expression is deferred until it's absolutely required.

```
main = do
  let x = 5
  let y = 3 + x
  let z = y^2+7
  print z
  print z
```

To print the result, the language finally forces evaluation of the expression.



Rather than computing the result, a graph is constructed which represents the eventual computation.

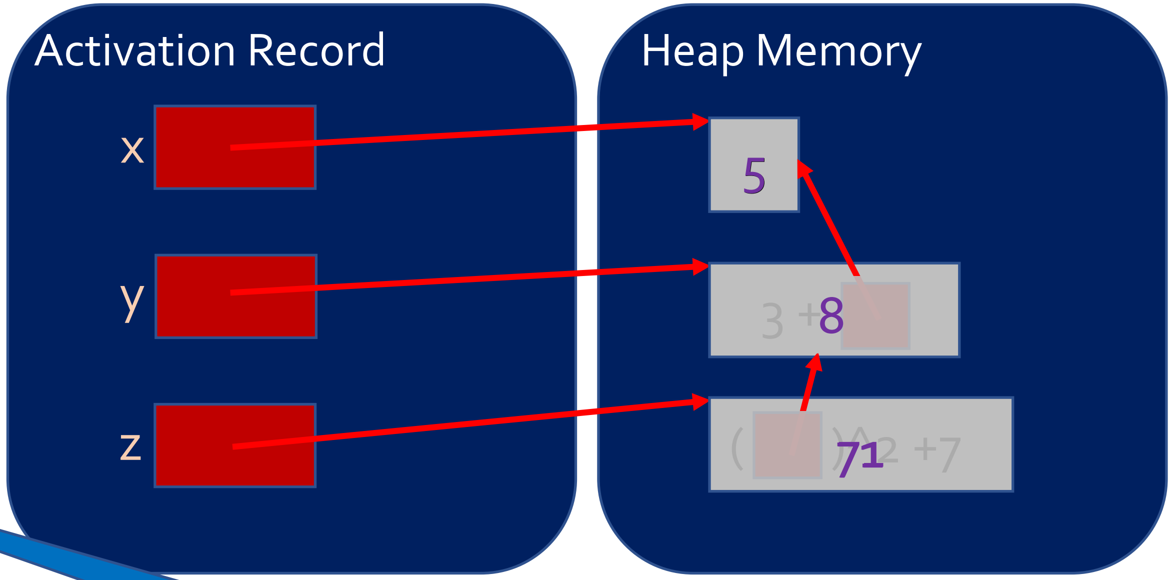
Every time you force evaluation of the variable, the expression is fully re-evaluated!

# Need Semantics

Need semantics works almost exactly like Name semantics!

The only difference is that the language memoizes (caches) the result of each evaluation to eliminate redundant computations.

```
main = do
  let x = 5
  let y = 3 + x
  let z = y^2+7
→ print z
  print z
```

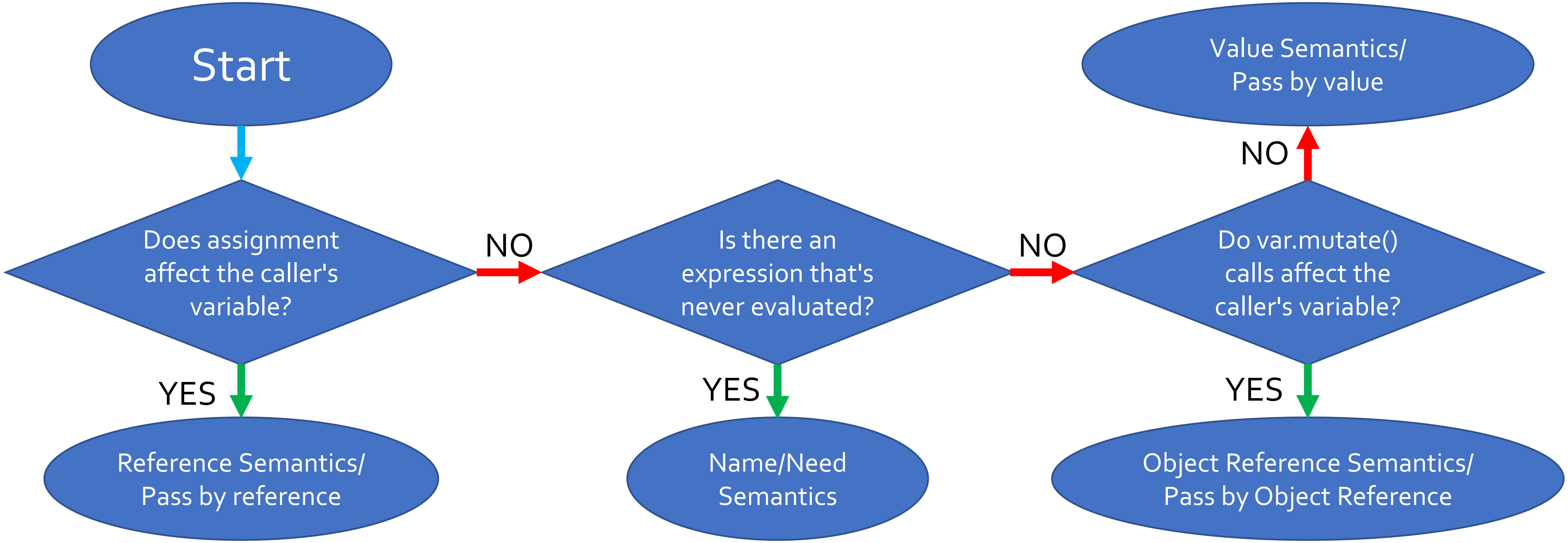


The first evaluation would compute and then memoize the result of the expression.

All later evaluations can just use the memoized result!

# Binding/Parameter Passing: How To Tell Which One

Imagine we give you a program and tell you its output…
How can you determine which binding strategy the language uses?



Start
Does assignment affect the caller's variable?
YES
Reference Semantics/ Pass by reference
NO
Is there an expression that's never evaluated?
YES
Name/Need Semantics
NO
Do var.mutate() calls affect the caller's variable?
YES
Object Reference Semantics/ Pass by Object Reference
NO
Value Semantics/ Pass by value

Inspired by former student Vincent Lin

# Classify That Language: Parameter Passing

```
procedure func1(v: Integer);
begin
  v := v + 3;
end;

function func2(var v: Integer): Integer;
begin
  v := v + 100;
  func1(v);
end;

var
  q, r: Integer;
begin
  q := 10;
  func2(q);
  writeln('q is ', q);
end.
```

Consider the following program, which prints:

```
q is 110
```

What parameter passing strategies is this language using?

# Classify That Language: Parameter Passing

```
procedure func1(v: Integer);
begin
  v := v + 3;
end;

function func2(var v: Integer): Integer;
begin
  v := v + 100;
  func1(v);
end;

var
  q, r: Integer;
begin
  q := 10;
  func2(q);
  writeln('q is ', q);
end.
```

This is how we define a pass-by-value parameter.

This is how we define a pass-by-reference parameter.

Consider the following program, which prints:

```
q is 110
```

at parameter passing strategies is this language using?

Answer: This language is using pass-by-value and pass-by-reference! This is Pascal!

# Human Interpreter: Binding Strategies

```
object Main extends App {
  def f(): Int = {
    println("Getting the value of x now!")
    1  // returns 1 as the result of f()
  }

  lazy val x = f()
  lazy val y = 3 + x
  lazy val z = y * y + 2
  println("About to print!")

  println(z)
  println(z)
}
```

The program to the left was written in a language that supports Need Semantics.

What does it print?

This is Scala!

# Human Interpreter: Binding Strategies

```
object Main extends App {
  def f(): Int = {
    println("Getting the value of x now!")
    1  // returns 1 as the result of f()
  }

  lazy val x = f()
  lazy val y = 3 + x
  lazy val z = y * y + 2
  println("About to print!")

  println(z)
  println(z)
}
```

All of these assignments are lazy, so their computation is deferred!

This is the first time we need the value of z, so this is when the computation happens.

Since this languages uses Need semantics, the values of x, y and z are cached so f() is not called again.

The program to the left was written in a language that supports Need Semantics.

What does it print?

Answer:

```
About to print!
Getting the value of x now!
18
18
```

This is Scala!