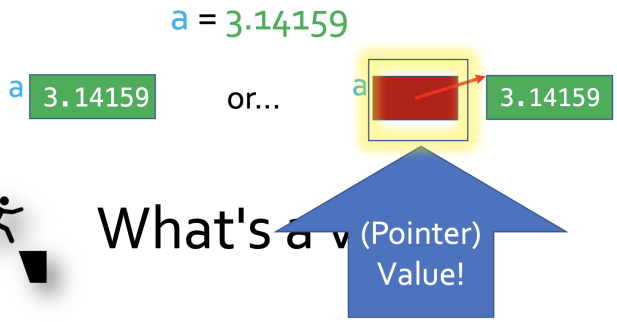
4 - Datapalooza

Variable vs. Value

What's a variable?

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



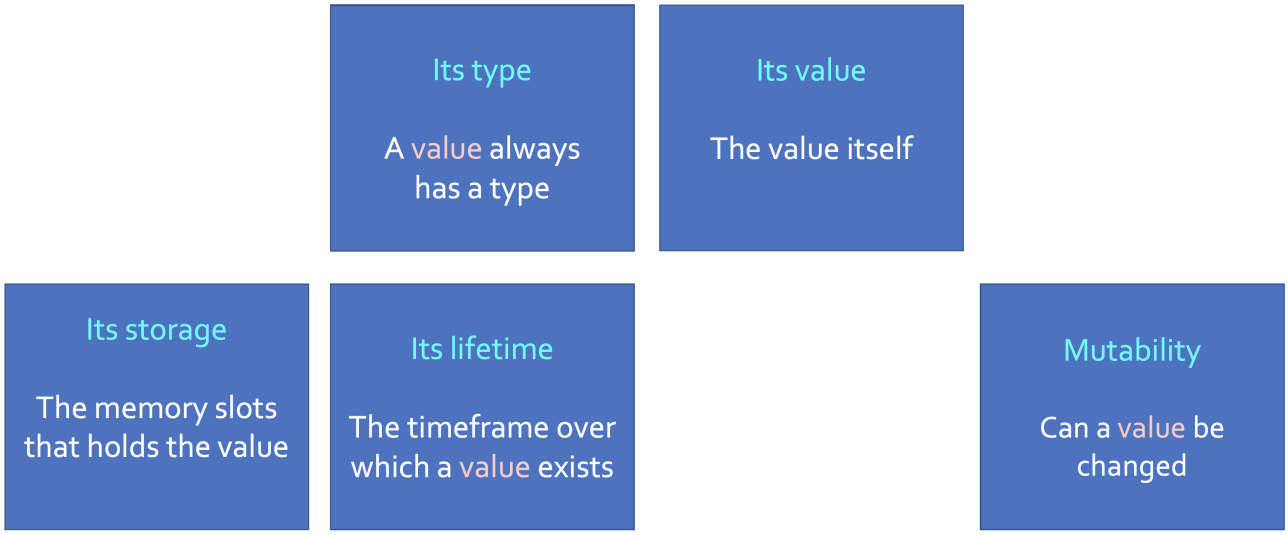
What's a v

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's in a variable

Its name How you refer to the variable	**Its type** A variable may (or may not) have an assigned type	**Its value** The value being stored and its type	**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 variable exists	**Its scope** When/where the variable name is visible to code	**Mutability** Can a variable's value be changed

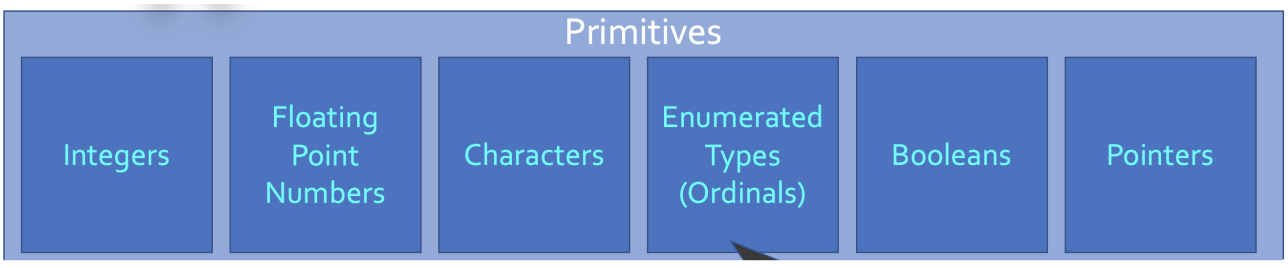
What's in a value



Types

- categorizes data, defines the size, encoding, operations, and casts i.e. how it's handled
- it is not necessary that all variables in a typed language (like Python, C, etc.) have types
  - in fact, no variables in Python have types
  - OTOH in Haskell, even though variables are dynamically typed, bc they are immutable, when they are bound, they are given fixed types
- i.e. a var is a type if it is perpetually bound to a type of value

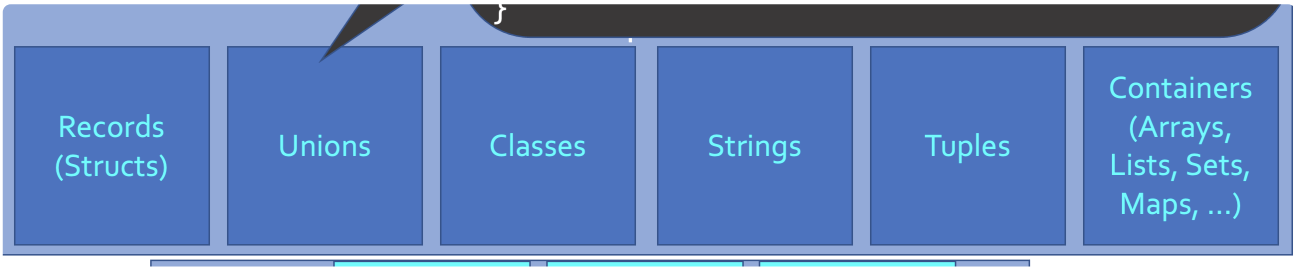
Primitives



- enum - an enumrated type with a set of possible values

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

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

Composites



- unions - can hold one of a few defined types

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
}
```

Other Types

- generic types - templated or type parameters

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;
};
```

- function types - functions stored as vars
- boxed types (mutable primitive) - an obj whose only value is a primitive

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_;
};
```

- user-defined types: the value the class defines (not the class itself), the value the struct defines, the value the enum defines
  - interface - list function that should be implemented (but not actual implementation) defines a new type

For example, every time you define a...

```
interface Drivable {
  void accel(int amt);
  void brake();
  void steer(int angle);
};
```

The language implicitly defines...

A type named Drivable

Value Types vs Reference Types

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_;
};
```

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

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_;
};
```

```
Shape s(Blue);  // Won't work!
```

If you recall from CS32, this defines an abstract method (aka "pure virtual").

It has no { code }, it's just a prototype.

This makes Shape an abstract class.

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

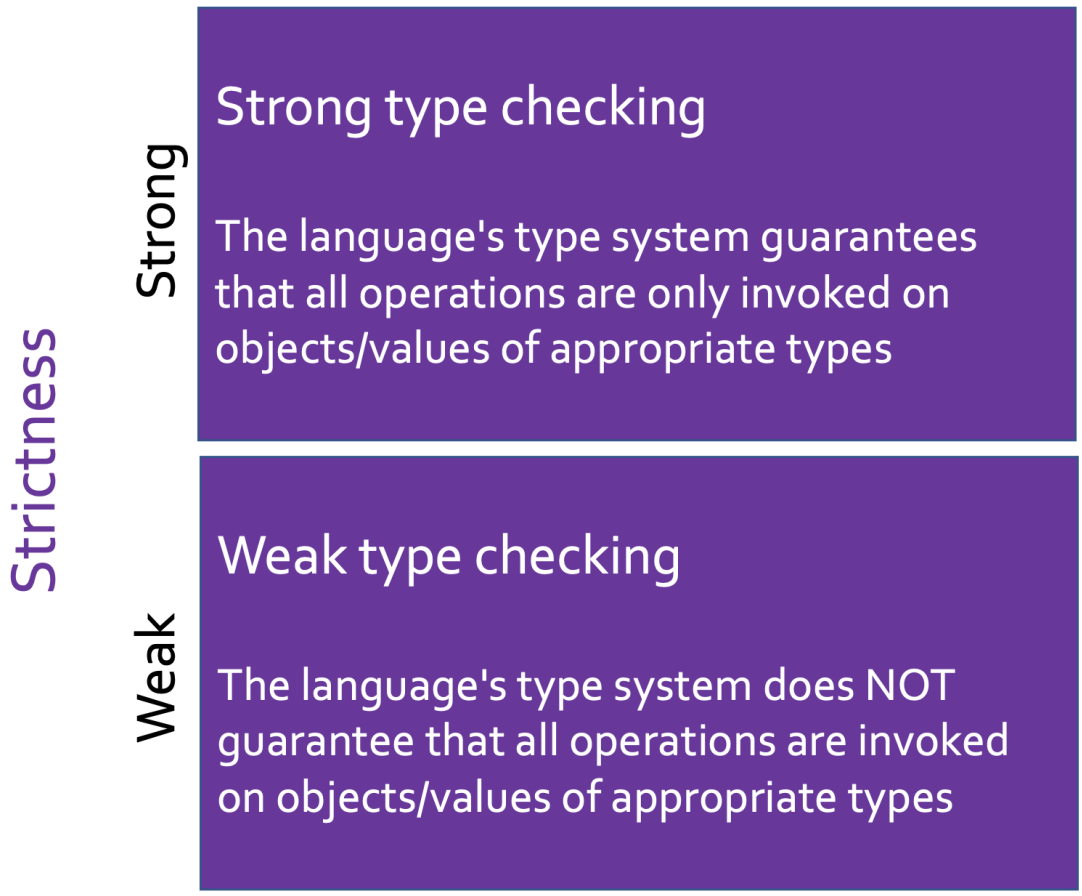
```
class Shape {
public:
  Shape(Color c) { color_ = c; }
  virtual double area() = 0;
private:
  Color color_;
};
```

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

Type Checking

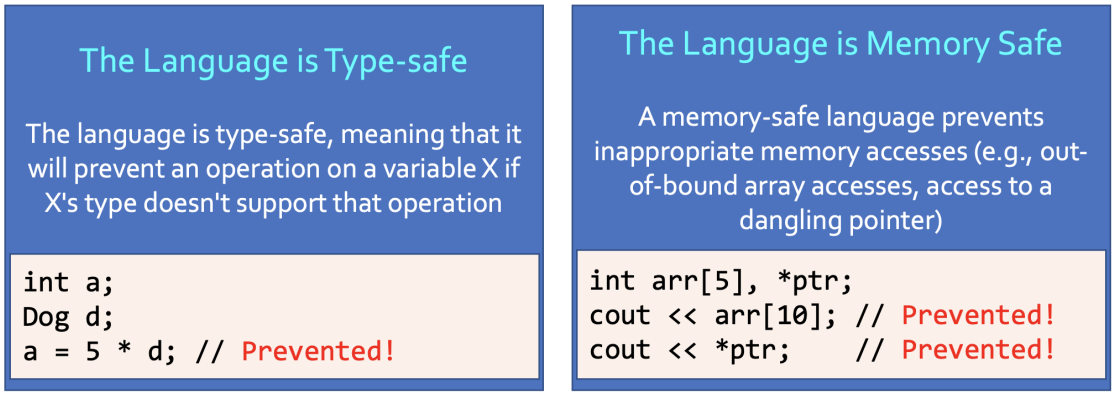
Strong vs Weak

- comparison

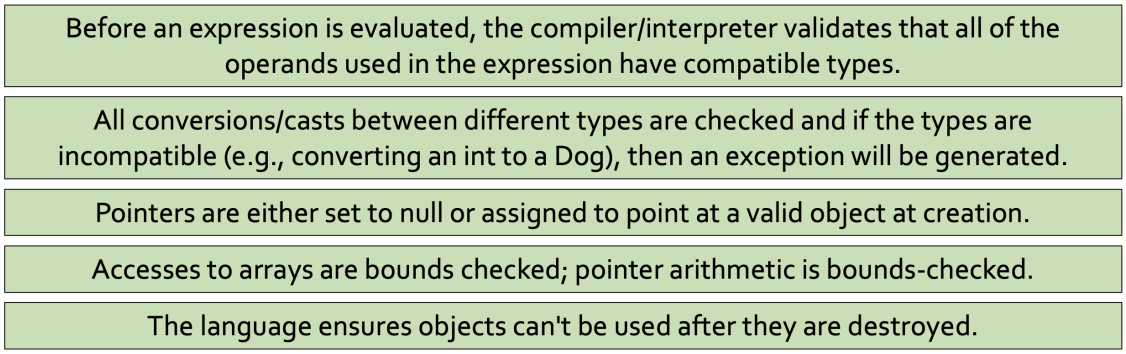


Strong

- guarantees operations invoked on **appropriate** types (doesn't have to be the same types)
- guarantees no undefined behavior due to **type-related** issues: type and memory safe



- strong type checking



- ensures memory-safeness because it needs to check the type of the undefined pointer or array out of bounds to validate operations
  - C/C++ are not strongly typed - they have undefined behavior for dangling pointers and out-of-bounds access
- Casting - uses checked casting to ensure casting to different values is ensured

```
// Strongly-typed Java has "checked" casts
public void petAnimal(Animal a) {
  a.pet();
  Dog d = (Dog)a; // Probably a dog, right?
  d.wagTail();    // It'll wag its tail!
}

...

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

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

- pros vs cons

Why Should We Prefer Strongly Typed Languages?

Dramatically-reduced software vulnerabilities (less hacking)

Earlier detection and fixing of bugs/errors

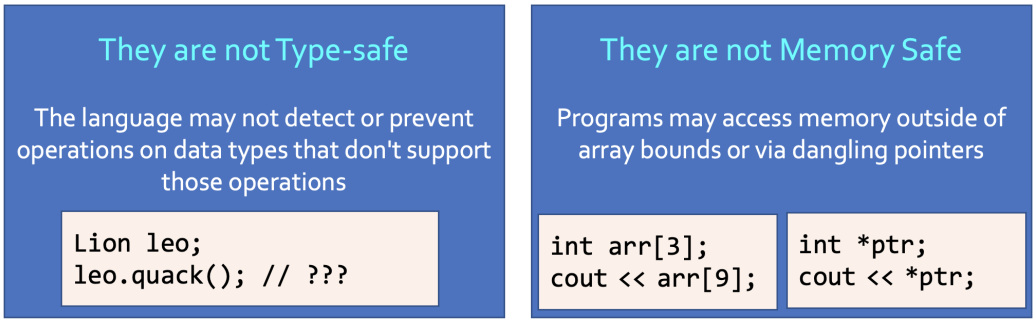
So Why Do People Still Use Weakly Typed Languages?

Performance and legacy.

Weak

- may cause undefined behavior due to type-related operations

Here are some attributes associated with weakly-typed languages:



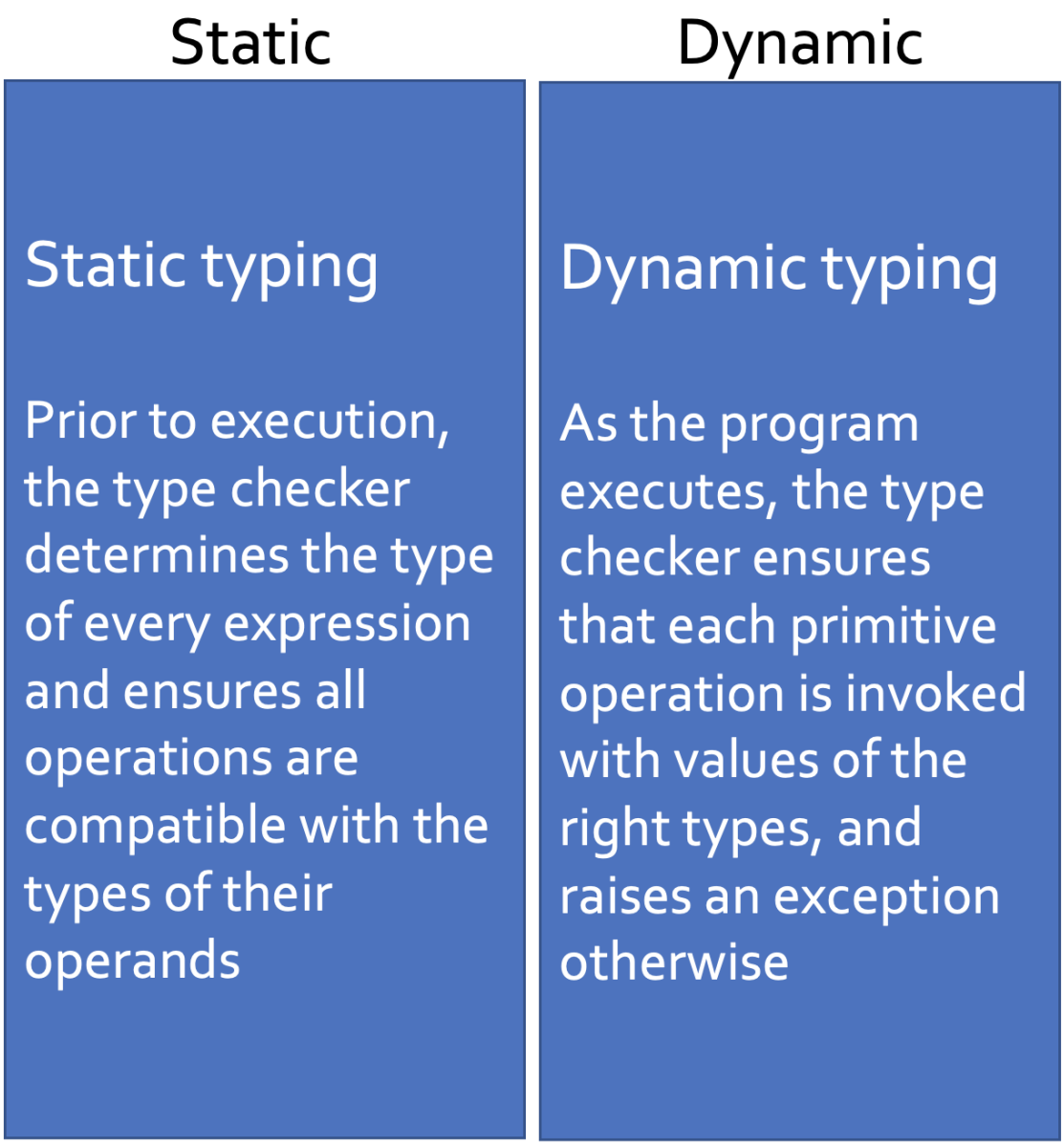
- weak languages allow unchecked casts → possible undefined behavior

```
// 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?!?!?
}
```

Static vs Dynamic

- static, dynamic, also gradual and hybrid typing, covered later

Compile-time vs. Run-time



Static

- even if the type is inferred, the value has a determined static non-mutable type, so it will check that all the params match the types specified

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

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

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.

- for static typing, the var must have a fixed immutable type

```
// C++

int main() {
  int a;
  a = 10;
}
```

```
# Python
def foo(x):
  if x > 5:
   a = 10
  else:
   a = "cats"
  ...
```

There's no way that a compiler could figure out the type of variable a when compiling this code.

- constraint satisfaction

Of course, it's never so simple!

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

Because we may discover conflicting evidence!

- Examples

```cpp
// 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;
  }
}
```

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

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

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

- Because static typing may need to infer, it is **conservative** and may prevent technically correct code from running if there is ambiguity in the types

```cpp
class Mammal {
public:
 virtual void makeNoise() { cout << "Breathe\n"; }
};
class Dog: public Mammal {
public:
 void makeNoise() override { cout << "Ruff\n"; }
 void bite() { cout << "Chomp\n"; }
};
class Cat: public Mammal {
public:
 void makeNoise() override { cout << "Meow!\n"; }
 void scratch() { cout << "Scrape!\n"; }
};

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

Static type checking can prevent technically correct programs from compiling!

Why? Because in order to guarantee type safety the type checker must be overly conservative.

Let's see!

```cpp
int main() {
 Dog spot;
 Cat meowmer;
 handlePet(spot, true, false);
 handlePet(meowmer, false, true);
}
```

  - in the above, bite/scratch will only call for dog or scratch, but the fact that mammal has no bite/scratch will say it is not the correct type (at compile type)
- pros and cons

Static Type Checking Pros and Cons

What are the pros of static type checking?

- Produces faster code (no type checks at runtime, optimizations possible)
- Detects bugs earlier in development
- 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

Dynamic

- types are associated with a **value**, not a variable; a variable is just a named binding
- uses secret type tags that are associated when a value is defined
- safety of operations is checked at run-time
- static may use dynamic during **down-casting**
  - down casting - casting a superclass to a subclass (e.g. person → student)
- pros and cons

Dynamic Type Checking Pros and Cons

What are the pros of dynamic type checking?

- Increased flexibility
- It's often easier to implement generics 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)

- must support ducktyping

```
class Duck {
public:
  void quack();
...
};

void say_hi(Duck &p) {
  p.quack();
}
```

```
class Duck:
  def quack(self):
    ...

class Banana:
  ...

def say_hi(p)
  p.quack()
```

- iff it quacks, then it s a duck (not tat only ducks can quack, so all functions can use all functions and is only checked at run time)

- supporting operations
  - iterable

```
# Python duck typing for iteration
class Cubes:
  def __init__(self, lower, upper):
    self.upper = upper
    self.n = lower
  def __iter__(self):
    return self
  def __next__(self):
    if self.n < self.upper:
      s = self.n ** 3
      self.n += 1
      return s
    else:
      raise StopIteration

for i in Cubes(1, 4):
  print(i)              # prints 1, 8, 27
```

- printable

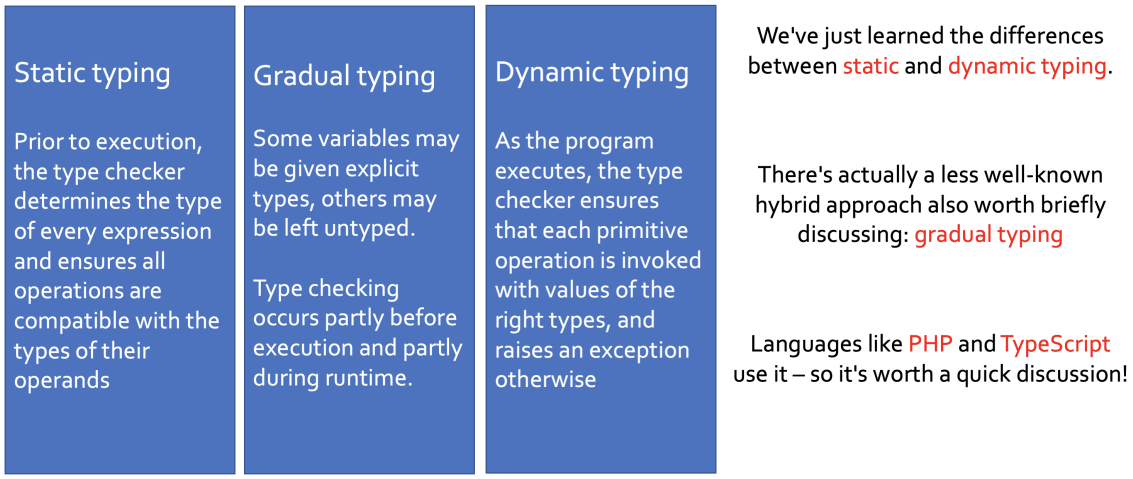
```
# Python duck typing for printing objects
class Duck:
 def __init__(self, name, feathers):
  self.name = name
  self.feathers = feathers

 def __str__(self):
  return self.name + " with " + \
         str(self.feathers) + " feathers."

d = Duck("Daffy", 3)
print(d)
```

- equality

```
# Python duck typing for equality
class Duck:
 def __init__(self, name, feathers):
  self.name = name
  self.feathers = feathers

 def __eq__(self, other):
  return (
    self.name == other.name and
    self.feathers == other.feathers
  )

duck1 = Duck("Carey",19)
duck2 = Duck("Carey",19)

if duck1 == duck2:
 print("Those are the same duck!")
```

Gradual Typing

- some variables may be given explicit types while others are untyped
- type checking occurs partly before execution and partly during runtime
- gradual vs static vs dynamic



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?

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

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

what_happens("foo")
```

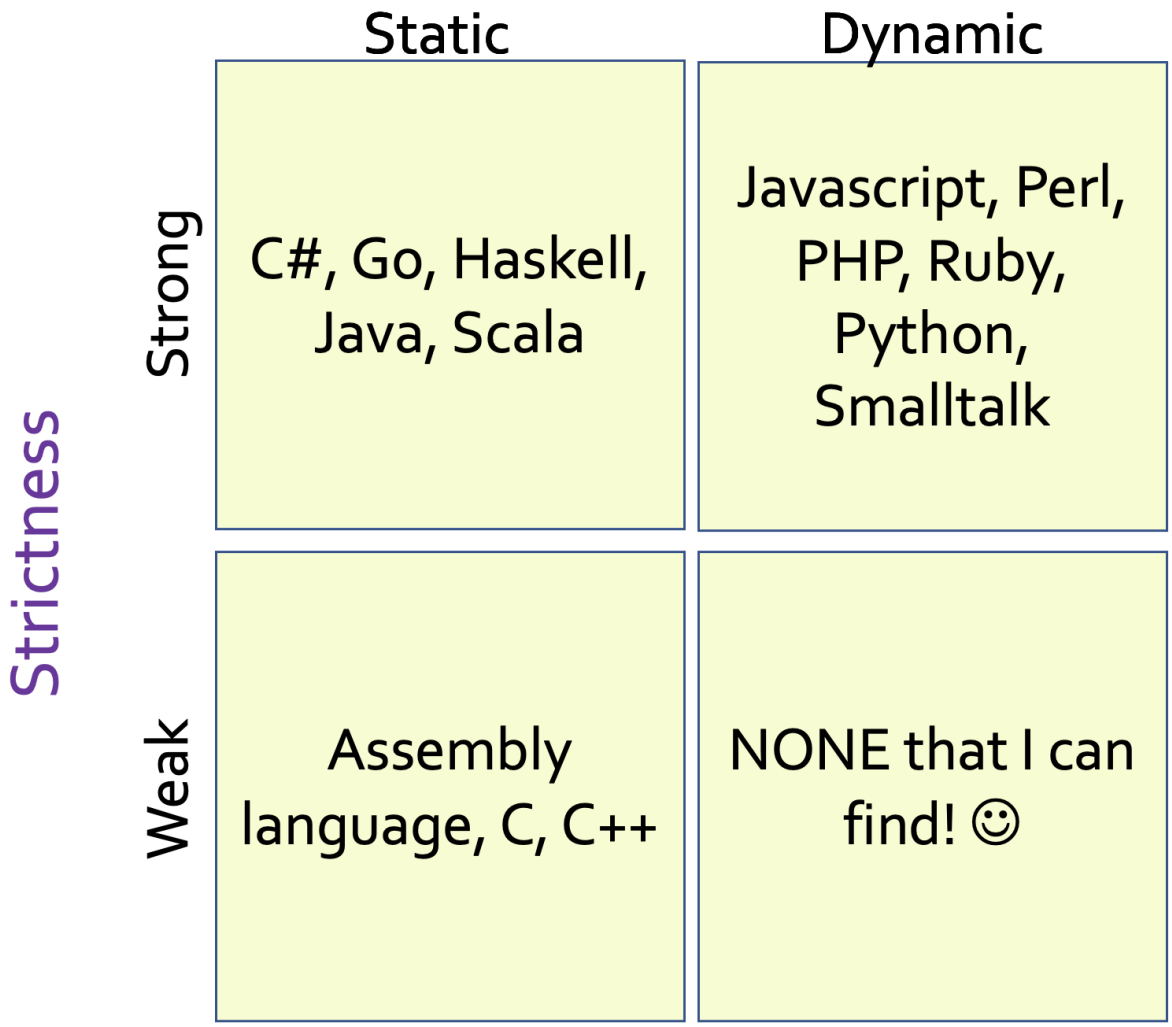
Challenge: Will a gradually typed language allow this? Why or why not?

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!

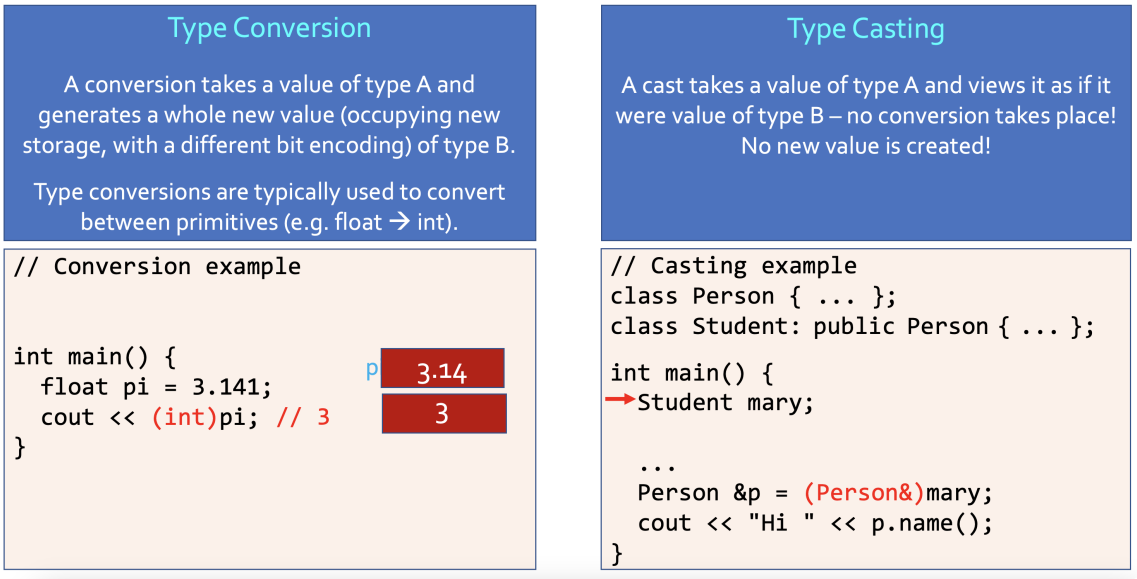
Language Examples

Compile-time vs. Run-time

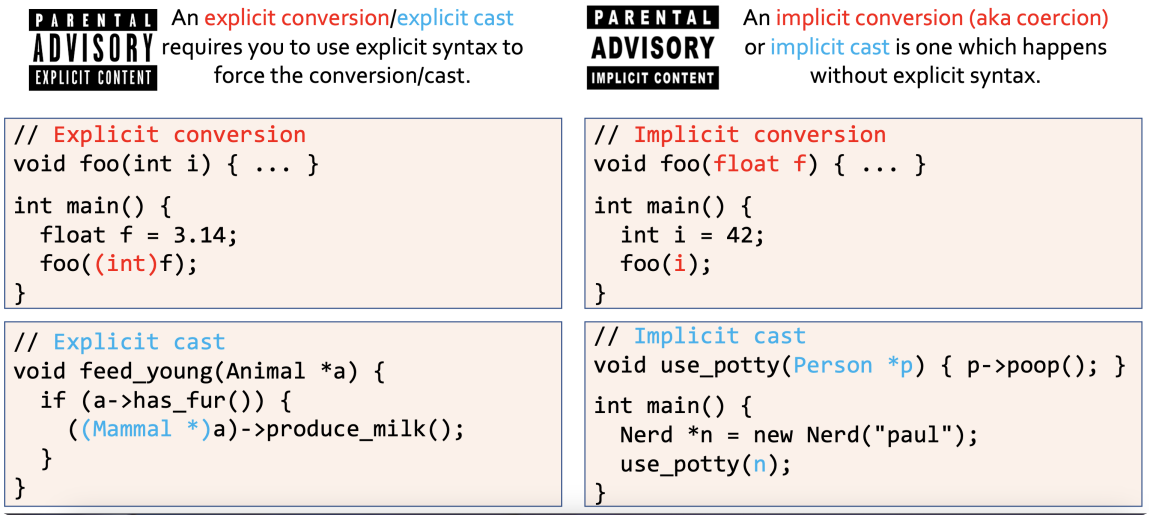


Type Conversion and Casts

- casting vs conversion
  - conversion creates a new object with the converted value
  - casting just treats the value as the new type without creating a new object



- implicit vs explicit cast/conversion
  - whether you need to explicitly write an expression to cast to a new class



  - upcast vs downcast
    - upcast casts to the superclass e.g., Nerd → Person
    - upcasts may be implicit
    - downcast is super to subclass e.g. Animal → Mammal
    - downcasts must be explicit
  - explicit casting tells the compiler to allow a compiler error to become a runtime check
    - e.g., if we didn't expollicitly cast the student to professor, do_your_thing would run a

give_a_lec which would be undefined error since studnets cant give lectures

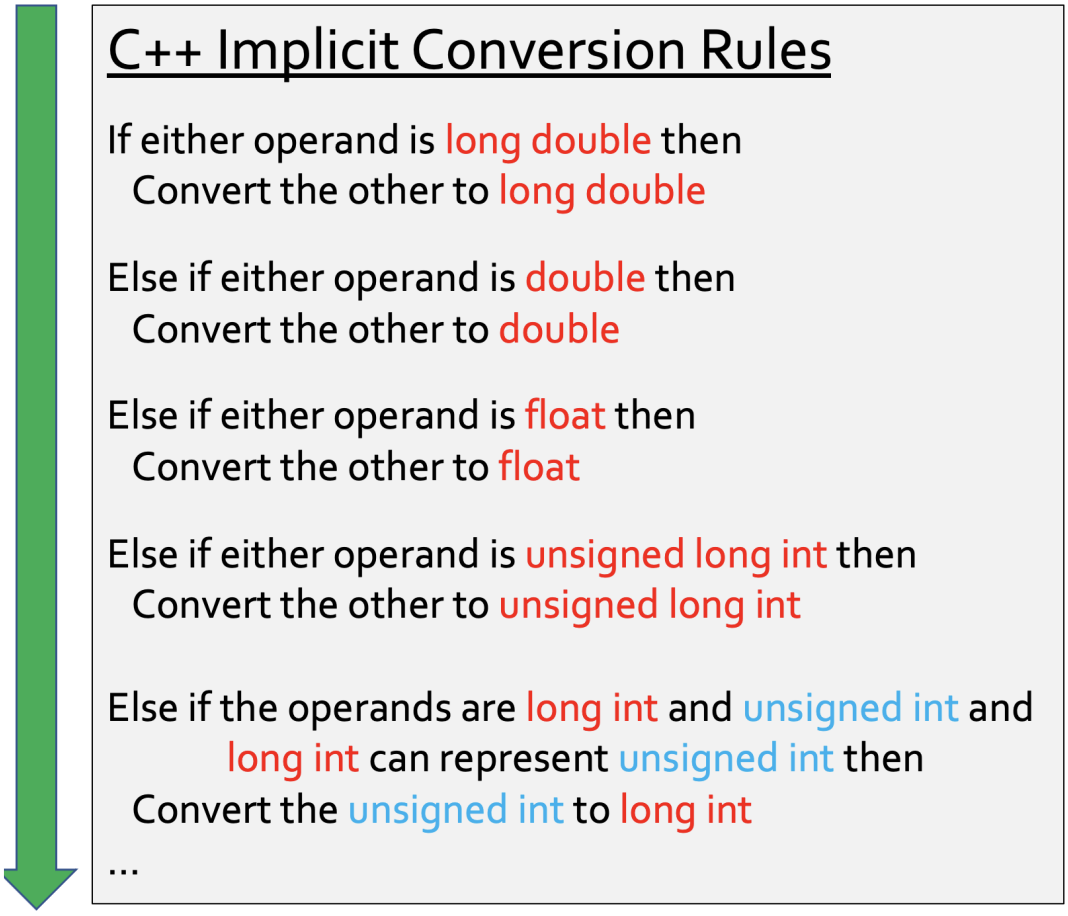
- but explicit casting will check to throw an error if its not possible instead of proceeding

```
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);
  }
}
```

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

Conversions

- coercions and promotions
  - coercions and promotions are **implicit conversions** ONLY → allows for operations even if types are not the same



- type promotions are conversions from a narrow type like `int` to `float` which s a wider type bc it can hold more possible values

- widening vs narrowing conversions
  - conversions can be narrowing or widening - narrowing or widening the range of possible values
    - ints are narrower than doubles
  - widening conversions are "value-preserving" since the values are guaranteed to be representable in the new conversion
  - narrowing is from wider to narrower OR if the range of values does not perfectly overlap e.g. float → int AND int→float are narrowing conversions and they not perfectly value-preserving in general

Casts

- widening vs narrowing casts
  - upcasts are widening conversions and downcasts are narrowing
  - because upcasts are guaranteed to work, they are always implicitly cast
  - downcasts are always explicit since they are not always safe

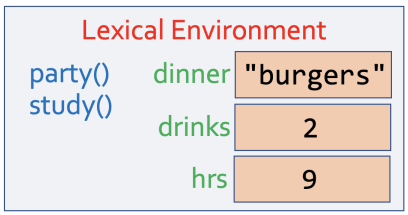
Scoping

- a variable is in scope if it can be explicitly referenced in that region of code
- a var may be alive but still out of scope
- scoping is associated with the name of the variable, so scoping checks for values bound to specific names
- the variable name that is currently in scope is actively bound to its name; anything else is inactively bound
- the list of actively bound variables is collectively the lexical environment

```
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);
}
```

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

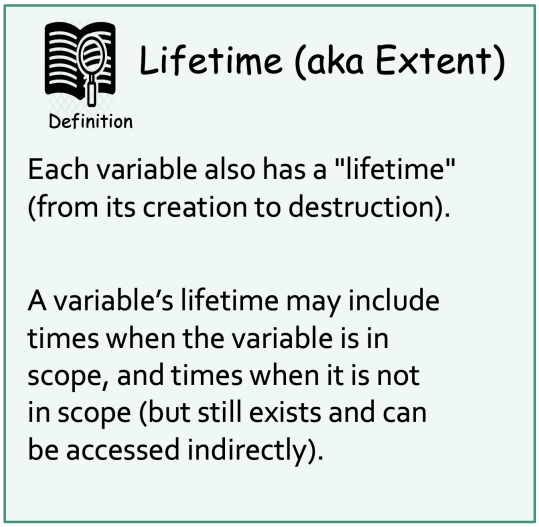
The set of in-scope variables and functions at a particular point in a program is called its lexical environment.



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

Lifetimes

- variable lifetimes



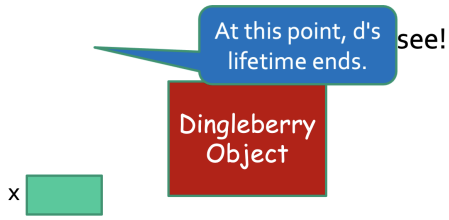

```
void study(int how_long) {
  while (how_long-- > 0)
    cout << "Study!\n";
  cout << "Partay!\n";
}

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

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

- value lifetimes

```
class Dingleberry:
 ...

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

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

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



- static lifetimes - variable/value exists till end of calling function, not just the function enclosing block

```
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

Lexical Scoping

- the usual scoping structure: sequentially nested

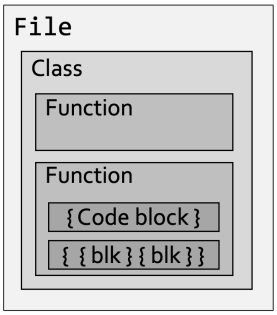
Let's start by discussing Lexical Scoping, which is by far the dominant scoping approach.

Lexical (aka Static) Scoping

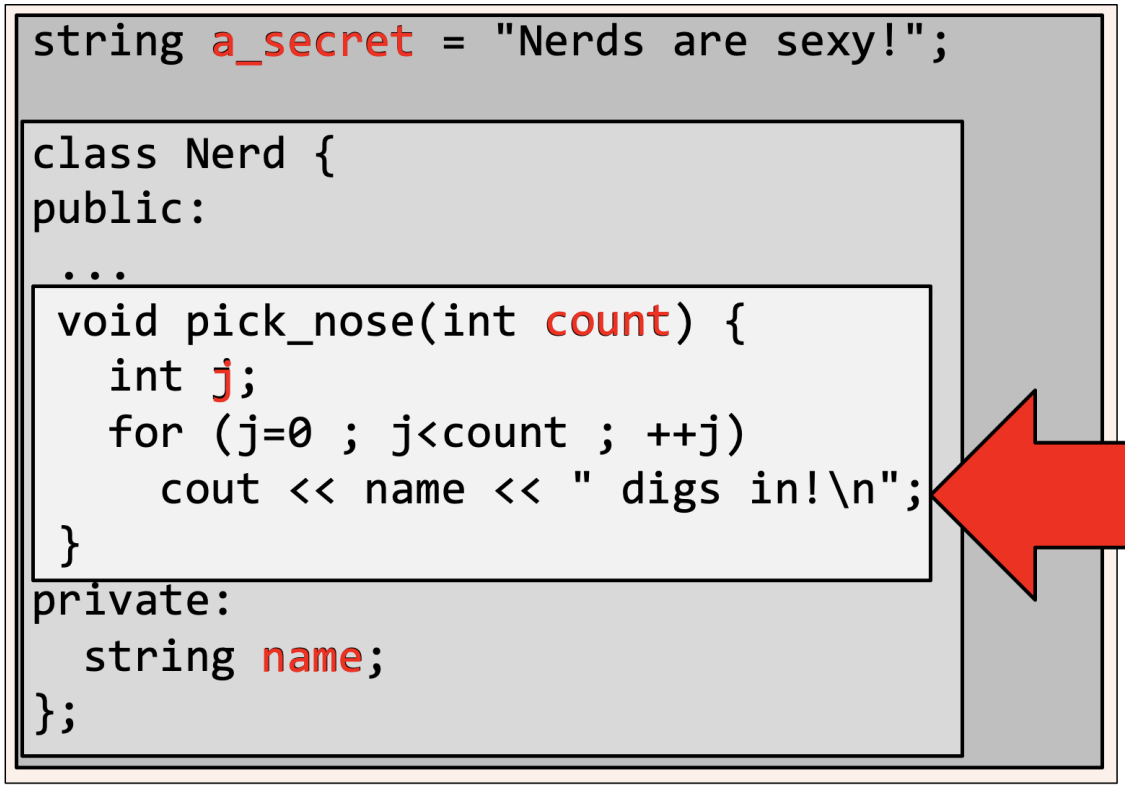
Definition

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!



- Python LEGB

```
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.

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.

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
```

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

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) …
}
```

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

- check-in current blocks, then in the enclosing block
- then searches the calling function and recursively backward until the main
- then the global scope

- If it is not found anywhere after checking the calling function, then no such variable
- FOR PROJ2: consider passing available variables as parameters to the function

Memory Safety

- memory safety refers to whether or not the language allows for behavior that may cause undefined behavior

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!

- Examples of memory-unsafe operations/langs

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();
```

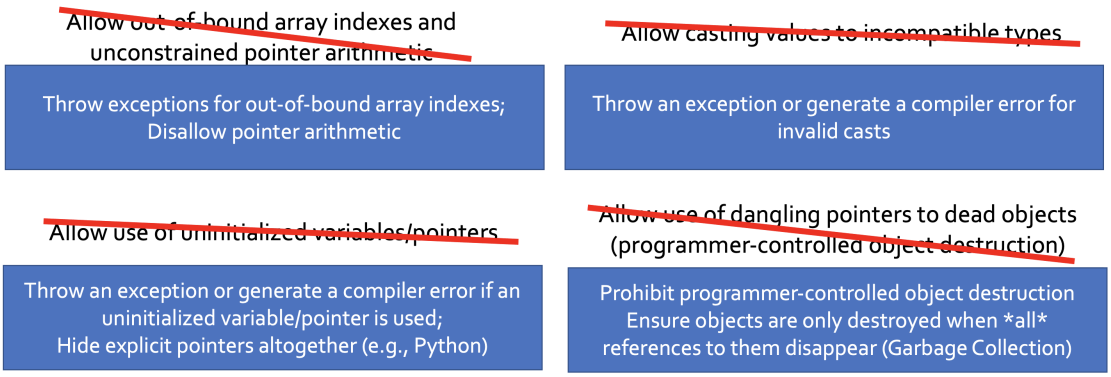
Allow use of uninitialized variables/pointers

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

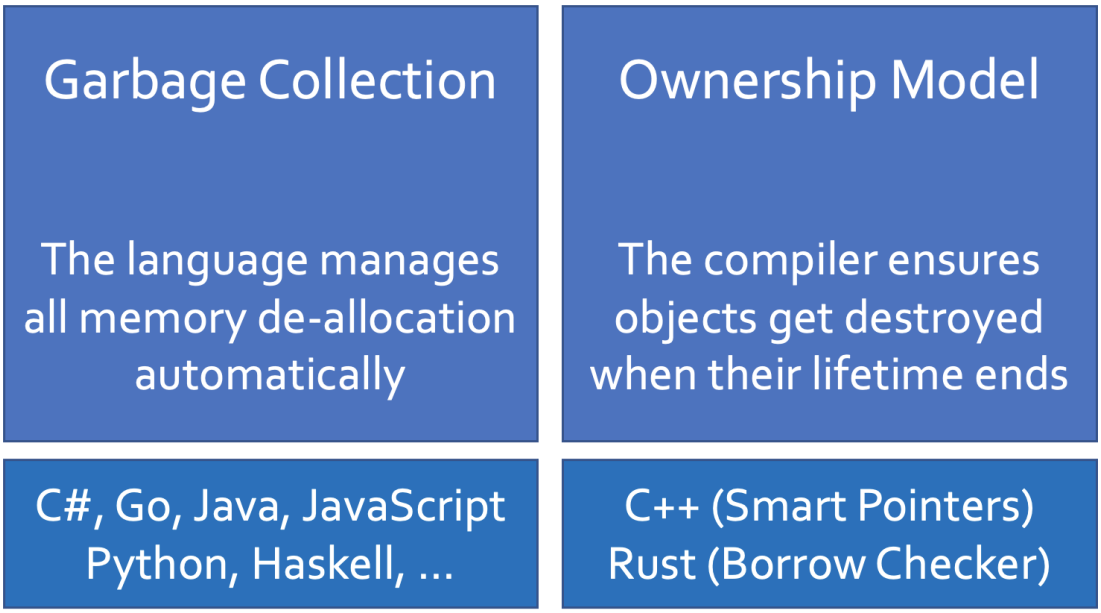
Allow use of dangling pointers to dead objects (programmer-controlled object destruction)

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

- Memory leaks are not necessarily memory-unsafe ONLY if it allows undefined behavior
  - languages may run out of memory or not have garbage collection that leaves memory leaks → but is still memory safe from an operations viewpoint
- Example of memory-safe langs/ops



- Memory management methods



Garbage Collection

- automated reclamation of allocated memory with no binding/reference

Eliminates Memory Leaks	Eliminates Dangling Pointers and Use of Dead Objects	Eliminates Double-free Bugs	Eliminates Manual Memory Management
Ensures memory allocated for objects is freed once it's no longer needed	Prevents access to objects after they have been de-allocated	Eliminates inadvertent attempts to free memory more than once	Simplifies code by eliminating manual deletion of memory

- a good rule is to garbage collect after each removal of reference (dangling)

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.

```
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;
}
```

- approaches

Mark and Sweep	Mark and Compact	Reference Counting
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.	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).	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.
Go, Java, JavaScript	C#, Haskell	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.