CS 131: Programming Languages (Fall 2022)

Rahul Reddy

Table of Contents

Lecture 1: Course Introduction

What's CS 131 About?

Languages come and go, but they are all based on paradigms and building-blocks.

Goals by Week 10:

- Given any new language, be able to groke how it works and write correct, efficient code in a matter of hours.
    - Our goal: Gain fluency in Python and basic proficiency with Haskell and Prolog
- Understanding of key programming language concepts
- Your ability to analyze languages you've never seen before
- Your ability to write correct code in our focus language

What is a Programming Language?

A structured system of communication designed to express computations in an abstract manner.

It can be used to define programs that specify the behavior of a computer.

- High level, "universal" languages are like C++ and python - that is those capable of implementing pretty much any valid program

What were the first high-level programming languages?

- First interpreted language: shortcode
- First compiled language: A-0

Why So Many Languages?

There are thousands of Domain-Specific Languages (DSLs) that folks have designed to accelerate problem-solving for specific use cases.

The Major Language Paradigms

- Imperative (C): What you're used to - programs are made of statements, loops, and mutable variables
- OOP Languages: Organized into classes and objects which send messages to each other to get things done
- Functional (Haskell): math-like functions. NO statements of iteration, only expressions, function, constants, and recursion

- Logic: programs define a set of facts: like(dogs, meat), category(beef, meat)

Language Choices

- Type Checking: static, dynamic, weak, strong, gradual
- Parameter passing: by-value, by-reference, by-pointer, by-object reference, by-name
- Scoping: lexical, dynamic
- Memory management: manual, automatic

    For example, Python: dynamic typing, by-object reference lexical, automatic memory management.

An important skill is the ability to logically deduce which design choices each language has made.

- Pass by object reference = Pass by pointer

Deep Dive Into Three Languages

Big-data approaches like Map-Reduce come from Functional Programming (FP).

What's needed to specify the details of a Programming Language?

Syntax Specification: Most Modern Languages use an encoding called Extended Backus-Naur Form (EBNF).

Semantic Spec: what the behaviors are (sometimes written in English)

What's a Compiler? What's a Linker?

- A **compiler** is a program that translates program source into object modules (which are either machine language, or bytecode targeted at an interpreter).
- A **linker** is a program that combines multiple object modules and libraries into a single executable file or library.

How Does a Compiler Work?

Source File → Lexical analyzer → Lexical Units → Parser → Abstract Syntax Tree → Semantic Analyzer.

The parser uses the language grammar spec to validate the tokens have a valid syntax.
If the syntax is valid, the parser converts the tokens into a tree that represents program's

operation.

It is converted into an Abstract Syntax Tree.

- The semantic analyzers check the semantic validity of the tree. It then updates the nodes of the tree and the output is an **annotated tree**, with type and other info added to the nodes of the tree.
- The **IR generator** produces an abstract representation of the program
- **Bytecode** is a binary encoding like a machine language but not for a real CPU

Front-end vs. Back-end

The compiler is encapsulated into two high-level components: Front-end vs Back-end

The front-end communicates to the back-end via the intermediate representation.

The Code Generator is the backend. It knows nothing about the original language - just how to generate machine code for the target platform.

We can easily generate machine code for many CPUs by replacing a single Component.

What language are most C++ compilers written in?

C++ through "bootstrapping" (Translator.c → C Compiler → Translator.exe)

If a C++ Compiler (written in C++) runs on x86 chips, how do you get it running on a new ARM chip with no existing compilers?

You cross-compile.

What's an Interpreter?

It is a program that directly executes program statements, without requiring them first to be compiled into machine language.

Load source file into RAM → (Optional: Convert program to an A or bytecode) → Initialize run-time state.

Introduction to Functional Programming

Haskell is one of the few pure Functional Programming languages.

Learn the principles of Functional Programming languages.

Learn to solve problems in a functional manner.

Understand the impact of Functional Programming languages.

Functional Programming & Haskell History

"Lambda Calculus" (developed by Alonzo Church) says any computable problem can be solved by composing one or more functions: e.g., $y = f(g(x))$.

Alan Turing proved that lambda calculus and Turing Machines are equivalent - both can compute the same sets of things in different ways:

- "The Church-Turing Hypothesis"

Lecture 2: Functional Programming and Haskell, Part 1

What is Functional Programming?

Every function must take an argument (f(x)) and must return a value (y = f(x))

Functions are "pure" and have no side effects.

Calling a function f(x) with the same input x always returns the same output y.

All variables are "immutable" and can never be modified.

Neural networks are partially functional programs!

Definition!

Pure Function: A function that has two properties

1) It does not depend on any data other than its inputs to compute a result
2) It doesn't modify any data beyond initializing local variables required to compute its output.

Functional vs. Imperative Programming

Imperative

Algorithmic - series of statements and variable changes

Changes in variable state are required.

Sequences of statements, loops, and function calls.

Multi-threading is tricky and prone to bugs.

The order code executes is important.

Functional

Transformation through function calls.

All "variables" are constants - no chances are allowed .

Function calls and recursion - no loops, no statements.

The order code executes is NOT important.

Haskell Background

Haskell is one of the few purely functional languages

The first version of Haskell didn't even have support for input or output.

- Getting input and output violates the lambda calculus

Haskell Tools

Haskell comes with a compiler and interpreter

Our First Haskell Program

hypot a b = sqrt ((a^2) + (b^2))

- Prelude is a haskell module that contains haskell definitions
- ":load" = load
- ":r" = reload

Haskell Data Types

Haskell is a "statically typed" language!

This means that the type of all variables and functions can be figured out at compile time!

- Python is dynamically typed which means that some types can ONLY be determined as the program actually runs.
- Haskell can also use "type inference" which means you don't have to specify variable/function types explicitly.

Haskell has a variety of built-in data types:

- Int: 64 bit signed integers
- Integer: arbitrary precision signed integers
- Bool
- Char
- Float
- Double

In python, integers are implemented as dynamic arrays of 32-bit numbers

Composite Data Types

- Tuple
- List
- String

List Of List Operations

- Head
- Tail
- Length
- Take

- Drop
- !!
- elem
- Sum
- Or, and
- zip

String Theory

- ++ = concatenation
- Haskell implements a string as a list of characters, so if we use “head” or “tail” on a string, it will output the single character or the rest of the characters in the string, respectively.

Constructing Lists: Ranges

- “range” feature
- “cycle” gives a never ending list so if we set tricycle = cycle [1, 3, 5], this would return an infinite list of [1, 3, 5, 1, 3, 5, 1, 3, 5, …] and the program would crash

Constructing Lists: Comprehensions

With a list comprehension, you specify the following inputs:

1) One or more input lists that you want to draw from (these are called "generators")
2) A set of filters on the input lists (these are called "guards")
3) A transformation applied to the inputs before items are added to the output list.

“if” is an expression in Haskell unlike in C++ where it is a statement

Comprehensions

Used also in C#, Python, and R, their primary use is for processing and creating lists using compact notation.

Intro to Functions in Haskell

1) Type information about the function’s parameter and return value
2) The unctinos’s name and parameter names
3) An expression that defines the function’s behavior (after the equals sign)

functionName :: String → String → String (Return Type)

```
functionName param1 param2 =
        BODY
```

Haskell uses spaces and indentation to define blocks of code

The Main "function" (IO Action)

```
main::IO()
main = do
        BODY
```

"Any_type" is called a "type variable"

It's Haskell's equivalent of a generic type in C++

Calling Functions

```
f :: Int → Int
f x = x^2

g :: Int → Int
g x = 3*x
```

⇒ "`Y = f g 2`" causes an error

Haskell evaluates function calls from left to right and as such, its function application is called "left-associative."

Mathematics	Haskell
f(x)	f x
f(x,y)	f x y
f(g(x))	f (g x)
f(x,g(y))	f x (g y)
f(x)g(y)	f x * g y

Lecture 3: Functional Programming and Haskell, Part 2

Let and Where Constructs: Local Bindings

Bindings in functions

Temporary variables are created with "let" and "where" constructs.

These constructs let you bind 1 or more temporary variables with expressions for use in functions.

Let:

```
get__nerd_status gpa study_hrs =
    let
        gpa_part = 1 / (4.01 - gpa)
        study_part = study_hrs * 10
        nerd_score = …
    in
        …
```

Where:

"Where" clause is very similar, but we put it at the end of the function.

When do we use let vs where?

When defining a variable for a single expression, you can use either.

When defining a variable for use across multiple expressions, use "where."

Let and Where Constructs: Nested Functions

We can also use "let" and "where" to define nested functions (in addition to local data bindings).

Control Flow in Functions:

If-Then-Else and Case Constructs

General Syntax of the if statement is:

```
if <expression> then <expression>
                else <expression>
```

```
ageist_greeting age =
        if age > 30 then "Hey boomer!"
                    else "Hey fam!"
```

Every if statement must have an else clause because every Haskell function must return a value

Case Statement:

```
case<expression> of
        const1 -> <expression>
        const2 -> <expression>
        …
        _  -> <expression>
```

(Just like in Haskell, every expression can be a function call, a variable, a constant, etc)

Guards – An alternative to If/Case

Function Guards are a unique functional language construct

A guard is like an if-then statement with a more compact syntax.

```
If <condition> Then <do-this>
        | <condition> = <do-this>
```

In Haskell, we can define a function as a series of one or more guards.

```
Somefunc param1 param2
        | <if-x-is-true> = <run-this>
        | <if-y-is-true> = <run-that>
        | <if-x-is-true> = <run-the-other>
        | otherwise = <run-this-otherwise>
```

In mathematical notation, we place the condition after the expression, but Haskell does the opposite.

Let's See Some Functions

```
- - Quicksort can be done in 8 lines!
Qsort lst
        | lst == [] = []
        | …
        …
```

More Control Flow: Pattern Matching

Pattern Matching is a feature of most functional programming languages which simplifies writing functions that process types and lists.

It does NOT introduce fundamental new capabilities to Haskell, but uses "syntactic sugar."

More Control Flow: Simple Pattern Matching

With Simple Pattern Matching, we define multiple versions of the same function.

Each version of the function must have the same number and types of arguments.

Each version of the function may specify required values for one or more of its arguments.

- When you call such a (group of) functions, Haskell calls the first version that matches the parameters passed in.

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n -1)

list_len :: [a] -> Int
list_len :: [] -> 0
list_len list = 1 + list_len (tail lst)
```

Pattern Matching with Tuples

For parameters that hold tuples, you can replace a parameter name with tuple notation and name individual fields in the tuple.

We can also use an **underscore** if we want to ignore a field in the tuple.

Pattern Matching with Lists

As with tuples, we replace the formal parameter of a function with a list pattern.

A list pattern is enclosed in () and has two or more variable names separated by colons:

```
(first: rest_of_list)
```

(first: second: rest_of_list)
(first: second: third: rest_of_list)

The first n-1 variables refer to the girls n-1 values in the list passed into the function.

Finally, we can use bracket and comma notation to match lists.

This second kind of list pattern is enclosed in [] and has one or more variable names constants or underscores separated by commas. Here are some examples:

[first]
[first, second]
[first, second, third]
[42]
[first, 19]

Every item matches a single value in a list, and will only match if the list is the exact length as the number of items in the pattern.

Advanced Functional Topics:

Haskell has "First-class" Functions

A first class function is treated like any other data.

It can be stored in variables, passed as arguments to other functions, returned as values by functions, and stored in data structures.

- It enables more efficient program decomposition and enable functions to generate new functions from scratch

Passing Functions As Arguments

```
talk_to :: String -> (String -> String) -> String
talk_to name talk_func
```

Returning Functions from Other Functions

Another aspect of first-class functions is being able to return the (and store them in variables)!

First-Class Functions

First-class and higher-order functions are a pillar of virtually all modern programming languages

Use cases include:

1) Providing comparison functions for sorting
2) As callbacks when events trigger
3) Multithreading (Executing threads concurrently/at the same time): in C++ ⇒

```
void foo() { //does some work and returns }
void bar(int x) { //does some work and returns }

int main() {
    std::thread thread1(foo), thread2(bar, 42);

    //run main, foo, and bar until they all finish
    thread1.join(); //pause until foo() finishes
    thread2.join(); //pause until bar() finishes
    std::cout << "main, foo, and bar completed.\n";
}
```

A Fundamental Set of Higher-Order Functions

1) **Mappers**: Perform a 1-to-1 transformation from 1 list of values to another list of values using a transform function
2) **Filters**: filters out items from 1 list of values using a predicate function to produce another list of values
3) **Reducers**: operates on a list of values and collapses them into a single output values

High-order Functions: map

A mapper is a function that maps a list of values to another list of values.

"map" in Haskell accepts two parameters:

1) A function to apply to every element of a list
2) A list to operate on

Type signature of a map is `map :: (a -> b) -> [a] -> [b]`

High-order Functions: filter

A filter is a function that filters items from an input list to produce a new output list.

"filter" in Haskell accepts two parameters:

- A function that determines if an intel in the input list should be included in the output list
- A list to operate on

Type signature of a map is filter :: (a -> Bool) -> [a] -> [a]

High-order Functions: reduce

A reducer is a function that combines the values in an input list to produce a single output value.

Each “reducer” in Haskell takes three parameters.

Lecture 4: Functional Programming and Haskell, Part 3

Reducer Functions: foldl

```
foldl(f, initial_accum, lst):
        accum = initial_accum;
        for each x in lst:
                accum = f(accum, x)
        return accum;
```

(**foldl** is left-associative – front to back)

Reducer Functions: foldr

(foldr is right-associative – back to front)

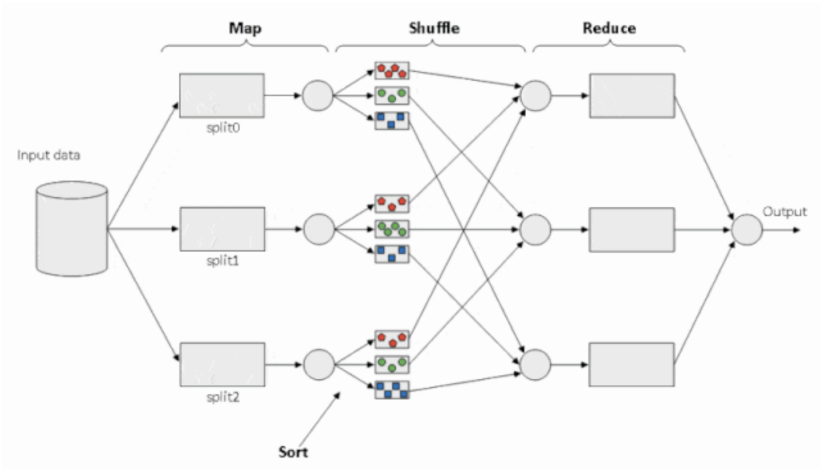
- In languages with lazy evaluation (like Haskell), **foldr** can be used to process infinite lists whereas **foldl** will crash

Map-Reduce Paradigm

This is used in medicine, autonomous vehicles, and other fields.

We can run these functions across thousands of servers simultaneously.

- A use case could be: to see how many times a URL was clicked on.



There are often multiple map steps and reduce steps performed in series (map1 → map2 → map3, etc). This is an example of the **divide and conquer approach**.

Advanced Topics in Functional Programming

Lambda Functions
Currying
Partial Application
Algebraic Data Types
Immutable Data Structures

> How can we add an item to a binary search tree (BST) in a functional languages where we can not mutate a data structure?

Lambda Functions

This is just like any other function, but it does NOT have a function name.
Normal Function: cube x = x^3 VS. **Lambda Function**: \x -> x^3

We use lambdas in higher-order functions when we don't want to bother defining a whole new named function.

> Lambdas let us define simple functions right where they're used to make our functions easier to read.

Defining a Lambda Function & Passing it to Another Function

Syntax:
\param_1 … param_n -> expression

- Anywhere we can pass a function by name to another function, we can pass a lambda!

Using a Lambda Function to Generate a New Function

```
wrapFuncWithAbs func =
    (\x -> abs (func x))
```

- This defines a function that returns another function as its return value

Closure

A closure is a combination of two things:

1) A function of zero or more arguments that we wish to run at some point in the future
2) A list of "free" variables + their values that were captured at the time the closure was created.

Lambdas & Closures

Lambdas and closures are used in most languages when we want to pass a simple function to another function.

One use case of these is to enable multithreading:

```
// Java thread example
public class LambdaThreadExample {
   public static void main(String args[]) {
    final int count = 50000;
    Thread t = new Thread(() -> {
      for(int i=0; i < count; i++)
         System.out.println("Child Thread: "+ i); });
   t.start(); // Start the background thread
   // Main Thread
   for(int j=0; j < 100000; j++)
      System.out.println("Main Thread: "+ j);
}
```

Up above, the () signifies the definition of a Lambda function in Java using a simple set of parentheses with 0 or more parameters inside.

We, then, provide the body: {}.

In this lambda, we have a **closure** which captures the variable **count**.

Currying

This transforms a function of multiple arguments to a series of functions that take a single argument each.

It converts a function of multiple arguments to a function that takes the first augment… which returns a function that takes the second argument… which returns a function that takes the third argument.. etc.. etc..

- Finally, the innermost function does the original computation.

Every function with 2+ parameters can be represented in curried form.

You can represent any function that takes multiple arguments by another that takes a single argument and returns a new function that takes the next argument.

Algebraic Data Types (ADTs)

These are like a combination of C++ structs and enums.

In functional languages, we use **pattern matching** to process algebraic variables.

- data Shape = Shapeless | Circle Double Double Double | Rectangle Double Double Double Double

Algebraic Data Types: Create a Tree

Let's examine a more interesting use of ADTs – creating a binary search tree

We'll start by looking at the definition:

```
data Tree =
        Nil |
        Node String Tree Tree
```

(An empty tree would point to NIL)

Now, let's see how to search our tree. We'll use a classic recursive search function and pattern matching.

Our search function will return **True** if it finds a value and **False** otherwise.

```
search Nil val = False
search (Node curval left right) val -- second pattern matches the Node variant.
        | val == curval = True
        | val < curval = search left val
        | otherwise = search right val
```

Algebraic Data Types: Adding a New Node to a Tree

This is not so easy because in functional languages, we can't modify existing values/variables.

We can create a new parent node for the new node we want to add (aka: We could create new node to replace the node we need to change)

- But now our new node (and its child) aren't part of our tree either.
- We can create a new node to replace the parent too and we have to do this all the way to the root.

Now, the code:

```
insert Nil val =
        Node val Nil Nil

insert (Node curval left right) val

        | val == curval =
                Node curval left right
```

```
| val < curval =
        Node curval (insert left val) right
```

So What's the Big O?

To add a new node, we need to generate replacement nodes: Best Case for BST: O(logN) – when the tree is balanced while the Worst Case for BST: O(N)

What Happens to the Old Nodes?

Garbage Collections automatically reclaims unused variables

Lecture 5: Python, Part 1

Objective: Understand Python well enough to write and code moderately complex programs

Python Uses
Quick & Dirty Scripting
Industrial Scripts
Web Backends (Flask, Django)
Data Science (Pandas, Numpy)
Machine Learning (PyTorch, TensorFlow)

Avoid Python
Writing Efficient programs – its interpreted
    But, it does have the ability to plug-in modules from other languages like C++ to optimize key operations! (Boost)

Python: An Open Source Language
The interpreter and libraries are written in plain old C.
    There's no python compiler - just a Python interpreter.
    - Most popular is CPython (3.9 or greater for this class).
    - The Python interpreter is called the Shell and it can be used interactively.

Code Formatting
No brackets for functions
No types for parameters
Single quotes and double quotes are the same

Program Execution
It is executed line by line (it's done top to bottom – synchronously).
    You can NOT put a function call before the function def.

Main function is just a regular function.
    It is conventional to have a function called main() in a Python program
You can run code outside of a function.

More Variables and Types

Python ignores **types** and it does not care (aka it does not type check).

We have them, so that code is more readable.

Python's type hints are used by a 'linter' which is a separate sort of compiler.

Lecture 6: Python, Part 2

Variables and Scope

In python, once you define a variable, it stays in scope until the function ends.

In contrast, in C++, variables go out of scope once the block ends (if statements, etc).

Looping with counters

For loops in python are not inclusive.

for i in range (2, 5) includes 2, 3, 4, but NOT 5.

break immediately terminates a loop (just like in C++).

Classes

They are quite different in Python compared to C++.

self is like this in C++.

A method has to have the self parameter as the first parameter.

This is how it's different from a normal python function.

- Also, you initialize member variables in the constructor (you do not need to define them)
- Every particular object in C++ has a fixed length of memory, but every object in Python could be different.
- In addition, a method is only accessible to an object of the class

```
c1 = Car(16) # Constructor is called when you initialize an object like this
```

In order to call another method within a method, we have to do: self.method.

Without the **self**, a variable will just be local.

self refers to a specific instance of a class.

A method can be public or private in python.

- A **double** _ makes a method private: __
- Python does NOT actually
- Most things are not immutable in python.

Allocation of Objects

Python does NOT have pointers, but it does have a similar concept called **object references**.

We use the dot operator with object references to call methods.

- In Python, ALL variables are object references (not just variables that refer to class objects).

If we do i = 42, then i = -10, it will first have the i value point to 42, then it will point to -10 and garbage collect -42.

This is very inefficient!

Every python object of a given class is NOT guaranteed to have the same set of member variables.

It depends on what **self** variables and code actually runs.

All integers are immutable.

Calling a Method

Always passing by object references (aka pointers), but never by value.

Copying of Objects

```
import copy
```

How to make a copy of objects:

```
c2 = copy.deepcopy(c)
```

Python: Automatic Garbage Collection

In Python, you do not have to worry about freeing objects when you're done with them (as opposed to C++).

Python automatically tracks each object and deletes it when your program no longer refers to it!

Some downsides to automatic memory management/garbage collection are that it is inefficient and slower. You have to track different variables and you have to track if a variable is being referred to.

Destructors

We would need to use a destructor if we did something like creating a temporary file in the file system.

Temporary files are not garbage collected.

- A destructor would be used to free a system resource (like a temporary disk file) that's not managed by the garbage collector.

Inheritance

A derived class inherits from a base class in the following way:

```
class Person:
    …

class Student(Person):
    …
```

A derived class method overrides a base class method by just redefining the method with the same name and parameters.

A derived method calls a base class method with super().methodName.

If you forget to call the base class's constructor, then it won't be called.

- When you call a method, Python figures out the right one for the particular object.

With polymorphism, you have to have a superclass and a subclass (base/derived class).

You must have a base class type variable.

Duck Typing

String Representation

Printing out an object (e.g., dogs, cats, etc being printed out in a certain way)

Enabling Classes to provide a description of themselves in a generic way
Enabling objects support comparison against other objects in a generic way
Enabling container objects to be iterated over in a generic way

Object Equality in Python

== checks if something points to the same object

is checks if something points the same value (if two object references refer to the same place in memory)

is not tested is two object references referrer to different objects in memory.

Object Identity in Python

Every distinct object has a unique ID number or "identity" in Python

id() function will print the memory address with this: print (id(a))

- Integers are immutable in python!

None

This is just like “null” and it is a way to represent nothing.

BUT, It is NOT a pointer value. It is an object

You can NOT check if a “None” value is false

Strings

In python, each string is an object just like our Circle objects

Strings are immutable, meaning they can’t be modified once created.

Python makes it super easy to extract substrings.

To extract a substring of length one from slot a, write str[a].

To slice a substring from slots[a-b), write: str[a:b].

- This would go from a (inclusive) to b (not inclusive).

Python provides lots of cool ways to process strings.

The “a in b” command returns true if a is a substring of b.

- There's also “a not in b.”

The strip command returns a new string with whitespace removed from the front/end of the string.

- The split command returns a list of all the tokens in a string, broken up by the specified delimiter.

Lists

Like strings, **lists** are **objects** in Python – and they support the same operations as strings!

But… unlike strings, lists are **mutable**!

- Lists can also hold values of all types.
- To concatenate to the end of a list: list.append(item) or list += [‘string’]
- Also, assignment creates a whole new list object!

Tuples

Python tuples are immutable, ordered, groups of items.

Tuples are commonly used for returning multiple values from a function.

Sets

Python has built-in support for the set ADT!

Python sets maintain a single unique copy of values.

C++ sets are ordered (implemented like a Binary Search Tree), but Python's are unordered (implemented as a hash table).

- This means that most operations are O(1): insertion, deletion, searching!

Python makes it easy to do things like set differencing, unions, and intersection.

\- = differencing
& = intersection
| = union

Dictionaries

Python has first-class support for dictionaries (maps) – they're super fast!

Unlike C++, Python dictionaries are unordered and you can NOT have a key more than once in the dictionary, mapping to multiple values.

Parameter Passing

C++ passes parameters **by value**, **by reference**, or **by pointer**.

Python does it, but only with one approach and its called **pass by object reference** (and it's identical to passing by pointer in C++).

- Parameters always refer to the original value.
- If you change a string inside a function it is not going to change the original string outside the function
- **append** (or +=) is a **mutating** function of **lists**.

Handling Errors in Python

When Python encounters an error that it doesn't know how to handle, it generates a special error called an "**exception**."

If you don't add code to "handle" an **exception**, it will cause the program to terminate.

- The function that **generated the exception** will immediately return, then the function that called it will immediately return, and so on, until your program exits!

However, it's easy to "handle" exceptions if we want to.

We can add:

```
try:
        …
except:
        # deal with exception here
```

```
        # in a graceful manner
        return None # Invalid result
```

Creating Larger Programs with Modules:

What are Scripts and Modules?

A **script** is a .py file that implements a main() function and is meant to run a stand-alone program.

Scripts are run from the command line, like this: python3 script_name.py.

A **module** is a .py file that implements a set of related classes or functions for use as a library (e.g., for machine learning).

Modules are intended to be imported into a python script or other modules to provide needed functionality.

Importing Modules

You can import a module and use its function/classes in your program.

To import the entire contents of a module, do this:

```
import math

def hypot(a, b):
    return math.sqrt(a**2 + b**2)
```

To import specific items from a module, do this:

```
from math import sqrt, cos, sin

def hypot(a, b):
    return sqrt(a**2 + b**2)
```

Creating a Module

To import a file foo.py: import foo

Difference Between a Module and a Script

A Python **script** is a .py file that's run from the command line,

A Python **module** is a .py file that is imported by another .py file.

Functional Influences: Comprehensions

Python has comprehensions just like Haskell and the syntax is pretty similar.

- We can create lists, sets, and dictionaries!

Functional Influences: Lambdas

Each lambda's body is a single expression on a single line and may refer to the lambda's parameters or variables from the current scope.

lambda p: p + ' has earwax'

Functional Influences: Map, Filter, Reduce

Map takes two inputs:

1) a function that accepts a single argument and returns f(arg) and
2) an iterable object of items – a list, set, dictionary, etc

Filter takes two inputs:

1) a function f() that accepts a single argument and returns True/False and
2) an iterable object of items – a list, set, dictionary, etc

Reduce takes two or three inputs:

1) a function f(accum, item): accepts 2 arguments and returns a new accumulated value.
2) an iterable object of items – a list, set, dictionary, etc (from functools import reduce – to use reduce).

Practical Python

Reading and Writing Text Files

To process a file, first we must "**open**" it. The open() function returns a file object which is an ADT that lets us read the contents of the file.

Reading from Files:

```
filename = '/Users/climberkip/foo.txt'
with open(filename) as file_object:
  file_contents = []
  for line in file_object:
    file_contents.append(line)
  # The "with" cmd auto-closes the file
# file_contents contains all the lines
```

"**with**" command automatically does this:

```
file_object = open(filename)
if file_object.was_opened_successfully():
   # do a bunch of processing
   file_object.close()
```

- If we wanted to do things more manually, we could do "file_object.readline()" which would read a single line from the file.

Writing to a text file on your hard drive:

Two Different Options:

1) Overwriting a File

```
filename = '/Users/climberkip/out.txt'
with open(filename, "w") as file_obj:
  file_obj.write('This deletes\n')
  file_obj.write('the original file\n')
  file_obj.write('and replaces it\n')
  file_obj.write('w/these 4 lines\
```

- "w" means we are opening the file for writing and will delete the original contents first.

2) Appending to a File

```
filename = '/Users/climberkip/out.txt'
with open(filename, "a") as file_obj:
  file_obj.write('This adds four\n')
  file_obj.write('lines to the end\n')
```

```
file_obj.write('of the original\n')
file_obj.write('file\n')
```

- "a" means we are appending data onto the end of the file, adding to its original contents.

Processing JSON Config Files

JSON is a *standardized* textual data format used to store and transmit information.

Common uses include transmitting records from a **server** to a **web browser** for rendering and storing configuration settings for an application.

- Each JSON "object" is enclosed in { bracers }.
- The object contains pairs of **key strings** and **values** (strings, ints/floats/booleans, Arrays of values, and Nested objects).
- JSON is an acronym for 'JavaScript Object Notation.'

Python provides built-in support for writing to and reading from JSON files!

Creating/Serializing a JSON File:

```
import json
course_data = {
        "courses":
        [{"name": "CS31", "Units": 4,
  "enrollment": "113/250"},
         {"name": "CS32", "Units": 4,
          enrollment": "180/200"},
         {"name": "CS131", "Units": 4,
          enrollment": "5/200"}
        ],
        "quarter": "fall"
}

with open("courses.json", "w") as file_obj:   # open "w" for writing the JSON to
        json.dump(course_data, file_obj)      # use the json module's dump() function to
                                                save our object in JSON format.
```

Loading/Deserializing a JSON File:

```
import json
with open("courses.json", "r") as file_obj    # open the desired JSON file for reading
        loaded data = json.load(file_obj)     # use the load function to load JSON data
```

```
                                                    and then the data is loaded as a dictionary
print(loaded_data[“quarter”])                       # prints: fall
print(loaded_data[“courses”][1][“name”)             # prints: CS 32
```

Processing a JSON String:

```
import json
json_str = get_json_weather_data('90024')
dict = json.loads(json_str)                         # use json.loads (load string) function to
                                                    convert the JSON string into a dictionary
print(f'The temperature is {dict["temp"]} degrees.')  # retrieve values from the dictionary
```

Practical Python: Connecting to Websites

Each URL consists of a scheme, a domain, a path and a query string:

```
https://www.google.com/search?q=silly+ucla+professors
```

- Typically the response will be an HTML page, but it's often JSON too!

web.py

```
import urllib.request # this module allows us to construct a web request
import urllib.parse

def get_web_Data(base_url, params = None):          # retrieve webpage or JSON payload
        if params is None:
          url = base_url
        else:
          param_str = urllib.parse.urlencode(params)
          url = base_url + '?' + param_str

        req = urllib.request.Request(url)
        with urllib.request.urlopen(req) as response:          # open connection to server
          web_page = response.read()          # read the data that was sent
          return web_page                     # return it
```

Pandas

Python's **pandas** module makes it easy to **analyze**, **plot**, and **clean** a dataset.

Analyze	Plot	Clean
Compute min, max, avg Compute percentiles Identify correlations in the data	Plot data from one or more cols Scatter-plot two columns Plot histograms for a column	Replace out-of-range values with reasonable replacements Eliminate rows with missing data

Pandas: Loading Data

We can import a dataset from a CSV (comma-separated value) file:

```
>>> import pandas as pd                 # rename the pandas module as "pd" for simplicity
>>> df = pd.read_csv('study.csv)        # loading a CSV file into pandas
                                        # read_csv() returns a "DataFrame" object which
                                        # holds the values from your CSV file

>>> print(df)                           # print data (omits some rows/columns to make it all fit)
>>> print(df.to_string())               # prints a full dump of the data
>>> df.describe()                       # pandas gives summary of the dataset (mean, min, etc)
>>> df.corr()           # shows us correlation between variables!
>>> import matplotlib                   # Python plotting library, inspired by MATLAB
>>> import matplotlib.pyplot as plt     # import to plot
>>> df.plot()                           # creates a plot
>>> plt.show()                          # shows plot on screen
```

- Plotting is done in two phases:
  1) Ask the dataframe to regenerate the plot (but not display it). – df.plot()
  2) Ask the matplotlib to display the plot on the screen. – plt.show()

```
>>> df.plot(kind = 'scatter', x = 'hrs_studied', y = 'exam_score')  # scatter plot: hours
                                                                    # studied to exam
                                                                    # score
>>> plt.show()                                  # shows plot on screen

>>> df["midterm_score"].plot(kind = 'hist')     # histogram plot for a particular field
>>> plt.show()                                  # shows plot on screen
```

Sometimes, your data will have gaps or spurious values.

For example, we might have missing data… or invalid values.

- Pandas will show missing values as NaN (Not a Number).

- We can also drop any row that is missing one or more values!

```
>>> df.dropna(inplace = True)      # if inplace is True, the dropna()
                                   # method mutates the current data
                                   # frame to remove all rows with
                                   # invalid values.
```

OR

- We can replace missing values with some default/average value.

```
>>> m = df["hrs_studied"].mean()
>>> df["hrs_studied"].fillna(m, inplace = True)
>>> print(df)
```

  - The **fillna()** method replaces all **NaN** values with your chosen value (mean, median, mode, etc)

- Finally, you can identify out-of-range values and replace them with reasonable replacements.

```
>>> for row in df.index:
...     if df.loc[row, "hws_completed"] > 10:
...         df.loc[row, "hws_completed"] = 5
>>> print(df)
```

  - For a dataset of n rows, df.index returns a Python range(0, n) that lets us iterate over the dataset's rows.
  - We can then use df.loc[row] to access the data at a particular row of the dataset.
  - So now we can replace all values that are invalid with a replacement (e.g., the mean).

<u>Integrating with C++</u>

Python is an interpreted language, so it's pretty slow yet it's the primary tool for tasks like **machine learning** and **data science**, which require *very* efficient computation.

To accomplish this, Python has the ability to import and use functions and classes written in **C++**!

- We can create a C++ module that can be used in Python – this is how **Tensorflow**, **Numpy**, or **Pandas** work!

Step 1: Install Python.boost

Step 2: Write your C++ classes.functions and register them with Python.boost

shapes.cpp

```
#include <boost/python.hpp>
using namespace boost::python;

class Circle {
public:
  Circle(double rad)
    { this->rad = rad; }

  double area() { return 3.14159 * rad * rad; }
  double circum() { return 2 * 3.14159 * rad; }
private:
  double rad;
};

BOOST_PYTHON_MODULE(shapes) {
  class_<Circle>("Circle", init<double>())
        .def("area",   &Circle::area)
        .def("circum", &Circle::circum);
}
```

- "shapes" is the module name, so for module shapes, we'd "import shapes".
- "init<double>" specifies that the Circle class constructor takes a single double as an argument
- Compilation/linking of boost C++ modules is pretty poorly documented!

Step 3: Compile your C++ code
Step 4: Use your C++ code in Python!

```
>>> import shapes
>>> c = shapes.Circle(10)
>>> print(f'The area is: {c.area()}')
314.159
```

Overall:

Compilation of code that uses the boost libraries is quite difficult and there are lots of command line options. Here's what worked:

- g++ -I/usr/include/python3.9 -fpic shapes.cpp -shared `python3.9-config --ldflags` -lboost_python39 -lpython3.9 -o shapes.so

With **boost**, **SWIG**, or **CLIF** you can create high-performance modules in C++ and import them in your Python programs!

This gives you the best of both worlds – Python's simplicity with C++'s speed!

Lecture 7: Data Palooza, Part 1

We're going to focus on how languages manage data (types, variables, & values).

Goal: pick up a new language and understand how it manages types, variables, & values.

What's a variable?

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

What is 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.

Variable Names

How variables are named:

Valid Characters
Case sensitivity
Maximum length
Reserved Keywords

- More important than naming rules are **naming conventions** - the standards enforced at big companies.

Variable and Value Storage

Stored in the **stack**, the **heap**, or the **static data area**

Stack – for local variables, parameters
Heap – for dynamic allocation (allocating on the heap is 20x slower than on the stack)
Static Data – Global Variables

Variable and Value Types

Based on a variable's type, we can determine:

It's type
Methods/operations we can call on it
Range of values it can hold
How to interpret the bytes stored in RAM
How values are converted between different types

Variable and Value Lifetime

Each variable and value has a lifetime over which it is valid and may be accessed.

A variable can be "alive" but not yet accessible by its name during the course of execution.

- Some languages give the programmer explicit control over a variable's lifetime

Variable Scope

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

There are two primary approaches for how languages address scope: lexical and dynamic.

Variable Bindings

As we've seen, different languages have different ways of associating (aka binding) variable names with values.

Variable and Value Mutability

Immutable means that its value can't be changed after its initial assignment.

There are different types of immutability (pointer, value, variables, etc).

- It seems less flexible, but it makes programs less buggy.

Types

In a typed language, Every variable does not have to have a type.

If a given variable is bound to a single value, then it has a type. Otherwise not!

- That said, a value is always associated with a type.

Primitives:

Integers, Floating Point Numbers, Characters, Enumerated Types (Ordinals), Booleans, Pointers

Composites:

Records (Structs), Unions (Variants), Objects, Arrays, Strings, Tuples, Containers (Lists, Sets, Maps, etc.), Generic Types, Function Types, Boxed Types

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

Type Checking Approaches

Strictness

Strong Typing: guarantees that all operations are only invoked on objects/values of appropriate types

Weak Typing: does NOT guarantee that all operations are invoked on objects/values of appropriate types

Compile-Time VS Run-Time

Static Typing: happens prior to execution (determines type of every expression,

Dynamic Typing: The type checker ensures each primitive operation is invoked w/values of the right types, and raises exceptions otherwise.

What is Static Typing?

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

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

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 NOT be changed.

Type Inference with Static Typing

Types do not need to be explicitly annotated for static typing. They can be inferred!

C++ offers a form of type inference, but is still statically typed.

- We can do this with the keyword "auto."

Static Type Checking is Conservative

It can prevent technically correct programs from compiling.

In order to guarantee type safety, the type checker must be overly conservative.

Dynamic Typing: Types Associated with Values!

Most Dynamic Typing languages don't require explicit type annotations for variables.

So a given variable name could refer to multiple values of different types over the course

of execution.

- Because of this, we say that with dynamic typing: “types are associated with values and not variables.”

How is Dynamic Type Checking Performed?

How does a program written in a dynamically typed language detect type violations at runtime?

The compiler/interpreter uses “**type tags**.”

- Type tags are an extra piece of data stored along with each value/object that indicates what type it is.

Dynamic Type Checking in Statically-Typed Languages

When we down-cast, we need to dynamically type check in statically-typed languages.

Down-casting is when we type cast from a superclass to a subclass (from, let’s say, a Person to a Student object).

Lecture 8: Data Palooza, Part 2

Duck Typing

It is a byproduct of dynamic typing.

Other Examples of Duck Typing

Ruby is a dynamically typed language and offers duck typing.

Javascript also has this, however it does not have classes – just objects.

A Hybrid Type Checking Approach: Gradual Typing

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

Type checking occurs partly before execution and partly during runtime.

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.

- If you specify a type, then the compiler will detect this and this will generate a compiler error.

If we pass an untyped variables to a typed variable, then the program will catch it at runtime but only if the types do NOT match.

What is a Strongly-typed Language?

This type of language ensures that we will NEVER have undefined behavior at run time due to type related issues.

There is no possibility of an unchecked runtime type error.

- The language is **type safe** and **memory safe** (it prevents inappropriate memory accesses).

Things We Expect in a Strongly Typed Language

Here are some techniques used to implement strong typing:

- Before an expression is evaluated, the compiler/interpreter validated that all of the operands used in the expression have compatible types.
- Pointers are either set to null or assigned to point at a valid object at creation,

- Language ensures objects can NOT be used after they are destroyed.
- Accessed to arrays are bounds checked.

General Principle: Avoiding operations on incompatible types or invalid memory

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

Checked Casts

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

Advantages of Strongly Typed Languages:

Reduced software vulnerabilities

Earlier detection of bugs

The Definition of Strong Typing is Strongly Disputed

These could be the definition of strongly typed, but are not necessarily true:

All conversions between different types must be explicit.

Language has to have explicit type annotation for each variable (int x = 5).

The type of each variable can be determined at compilation time.

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).

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

Type Relationships: Super-types and Sub-types

Given two Types, T_super and T_sub, we say:

T_sub is a subtype of T_super iff

An object of type T_sub can be used anywhere that code expect an object of type T_super, and Every element belonging to the set of value of type T_sub is also a member of the set of values of type T_super,

int is a **subtype** of long.

All operations that can be performed on a long can also be performed on an int.

Every int value is also in the set of long values.

A non-const variable is a subtype of a const variable.

Type Conversions and Type Casts

There are two ways to convert between types – **conversion** and **casting**.

Type Conversion

We generate a whole new value of the new type. The new values occupies separate storage and has a different bit encoding.

Type Casting

We reinterpret the bit of the original value in a manner consistent with the new type. No new value is created!

Conversions and Casts can be **widening** or **narrowing**.

Widening means the target type is a supertype and can fully represent the source type values (e.g., int → long, long → double).

Narrowing means the target type may lose precision and otherwise fail to represent the source type's values.

Conversions and Casts can be **explicit** or **implicit**.

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

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

Implicit (also called coercion) is one which happens without explicit conversion syntax.

Explicit Type Conversions: Boxing and Unboxing

A **Boxing** Conversion converts a primitive into an object that holds the primitive values as its only member.

An **Unboxing** Conversion converts a boxed object back to a primitive value.

Implicit Type Conversion Rules

Most languages have a set of rules that govern implicit conversions that may occur without warnings/errors.

Type Casting: Reinterpreting Bits

No new value is created by the cast – we just interpret the original bits differently.

The most common type of casting is from a **subtype** to a **supertype**.

- For example, **upcasting** from a subclass to a superclass to implement polymorphism
- In most languages, upcasting can be done without an explicit type cast.
- **Downcasting** is also used, where we cast from a **superclass** to a **subclass**.

  This also often requires an explicit type cast.

Pros and Cons of implicit vs. explicit conversions

Implicit Conversions:

Simpler, Less verbose code, Higher likelihood of bugs, and more convenient.

Lecture 9: Data Palooza, Part 3

Scope

This is associated with the range of a program's instructions where the variable is known.

Scope changes as a program runs.

- Another way to say that a variable is in scope is to say that it has an active binding.

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.

Another name for the **lexical environment** is **activation record**.

- Lexical Environment (unlike activation record) includes variables in the class surrounding the functions or global variables in the class. It also includes the nested function and is the overarching set of variables, methods, functions that are visible.

Lifetime

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

It may include times when the variable is in scope and when it is not (but still exists and can be accessed directly).

- Values also have lifetimes" and a value's lifetime might be different than the variable that refers to it.

Lexical Scoping

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

We determine all variables that are in scope at a preposition X in our by looking at X's context first, then looking in successively larger enclosing contexts around x.

- Most programs use this: Python, C++, C, etc

Lexical Scoping (Python)

Local, Enclosing, Global, and Built-in ("LEGB" rule).

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

Enclosing: Look in the enclosing function that contains your function

Global: Then look at all of the top level variables and function

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

Contexts for Lexical Scoping

Expressions, Classes/Structs, Namespaces, Global (Variables)

Lexical Scoping Tricks: C++

Sometimes we'll have code that defines different variables of the same name, but at different scopes. This is called **shadowing**.

Do NOT do this, but know how it works.

- To access a local variable, just use the variable name.
- To access member variables when there's already a local variable with the same name, use this->variable_name.

Lexical Scoping Tricks: Python

To change a global variable x, first use global x.

To access a class member variable x always use self.x.

To create a new local variable (even if it has the same name as a global): x = init_value.

Dynamic Scoping

As your program runs, it keeps track of every variable it defines in the order it defines them.

When you refer to a variable, the program finds the most recently defined variable of the same name and uses that as your variable.

Binding Strategies:

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 (C++).

Reference Semantics:

a variable name is directly bound to another variable's storage, like an alias (C++, C#).

Object Reference Semantics:

A variable name is bound to a pointer that points to an object's value (Java, Javascript, Python).

Name Semantics:

A variable name is bound to a pointer that points to an expression graph that can be evaluated to get a value (Haskell, R).

Reference Semantics

When a language does this, you may assign an alias name to an existing variable and read/write it through that alias.

Technically, most references are implemented using pointers under the hood.

Object Reference Semantics

When a language uses **object reference semantics**, your variable name is bound to a pointer variable that points to an object/value.

Pointers are explicit, and you can read/write them unlike **reference semantics** (which hides its use of pointers).

- Expressions that refer to a variable still use the object itself, however.

Counterintuitively, assignment of variables changes the **pointers** and not the pointed-to-objects.

Java also uses **object reference semantics** for all objects, but not for primitive types like ints and doubles.

This is the most dominant paradigm in most modern languages: C#, Java, Python, etc

Object Reference Semantics: Testing for Equality

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 are different objects in RAM)?

Pointers: Explicit Object References!

When you use pointers in C++, you're also using **object reference semantics**, but using explicit * and & syntax to define and read/write pointed-to data.

To summarize, a pointer is an explicitly defined object reference.

- If you omit the dereference operator (*), pointers are exactly like object references in python and Java.

Name Semantics

These languages bind each variable name to a pointer that points to an expression graph called a **thunk**.

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.

- Languages like Haskell implement a variant of this called **Need Semantics** which **memoize** (cache) the result of each evaluation to eliminate redundant computations.

Memory Safety

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

Memory-unsafe languages allow memory operations which lead to undefined behavior.

- They allow out of bound array indexes, unconstrained pointer arithmetic, casting values to incompatible types, and use of uninitialized variables/pointers.
- They also allow use of dangling pointers to dead objects (programmer-controlled object destruction).

Memory Safe Languages throw exceptions for out of bound array indexes and disallow pointer arithmetic.

- They throw an exception or generate a compiler error for invalid casts.
- They throw an exception or generate a compiler error if an uninitialized variable/pointer is used

  They hide explicit pointers altogether.
- They prohibit programmer-controlled object destruction (objects only go away when you are no longer referring to them).

Strategies for Memory Leaks and Dangling Pointers

Garbage Collection

The language manages all memory deallocation automatically.

Smart Pointers (important for industry if you want to work as a C++ Engineer)

- Libraries that enable the programmer to reduce memory leak bugs (used by C++).

When Should Objects be Garbage Collected?

We should "garbage-collect" an object when there are no longer any references to that object.

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

Garbage Collection Approaches

Three Main Approaches

Mark and Sweep

Discover active objects by doing a traversal from all variables that are object references. Free all objects that were not reached during discovery.

- Cons: If large amounts of RAM are processed, it can cause **thrashing** with **OS Paging**. In addition, it can also cause **memory fragmentation**.

Definition: *Thrashing* is the poor performance of a virtual memory (or paging) system when the same pages are being loaded repeatedly due to a lack of main memory to keep them in memory (happens when a computer's virtual memory resources are overused and causes the actual throughput of a system to degrade by multiple orders of magnitude).

Definition: *OS Paging* is a storage mechanism used to retrieve processes from secondary storage to the main memory as pages.

Definition: *Memory Fragmentation* is when your memory is allocated in a large number of non-sequential blocks with gaps that can't be used for new allocations due to size differences.

Mark and Compact (slower than **Mark and Sweep**!)

Discover all active objects and move them into a new block of memory. Then, throw away everything in the old block of memory (which holds only dead objects).

- Pros: Eliminates memory fragmentation

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.

Garbage Collection and (Un)Predictability

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

Objects do a lot more than just being tied to memory pressure. They can create temporary files, open network connections, and database server connections.

- If each object creates a large temporary file on the hard drive and if there's plenty of RAM, the collector does NOT run and gets rid of unreachable objects (and their temporary files) often.
- IN Garbage Collection based languages, the programmer really needs to free other resources manually and assure GC will NOT happen.

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

Reference Counting-based Garbage Collection

Every object has a hidden count to how many references there are to it.

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, then the object ends up being deleted.
- This is not thread-safe! Neither is python! We have to pause all the threads during garbage collection.

In most cases, Python will only run one thread at a time even if you create another thread.

When an object is destroyed, all objects transitively referenced by that object must also have their reference counts decreased!

When a certain object's reference count goes to zero, then the objects that it points to will have their reference counts also go to zero and then these will require them to be garbage collected as well.

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

Reference Counting-based Garbage Collection PROS and CONS

Pros: Simple, and efficient memory usage

Cons: Updating reference coins ust be done in a thread-safe way. SLOW!

C-Python is single threaded in this way

Garbage Collection eliminates entries classes or common memory safety bugs.

But, it adds extra storage and performance overhead.

Memory Safety: What Happens When Objects Die?

Many objects hold resources which need to be released when their lifetime ends. There are three ways this is handled:

Destructor Methods

Finalizer Methods

Manual Disposal Methods

Destructors

These are only used in manual memory management languages like C++.

Finalizers

These are used to release unmanaged resources like file handles or network connections, which are not garbage collected.

- Finalizers may cause **synchronization** issues, even in otherwise sequential (single-threaded) programs, as finalization may be done concurrently (concretely, in one or more separate threads.
- Finalizers may cause **deadlock** if **synchronization** mechanisms such as locks are used, due to not being run in a specified order and possibly being run concurrently.

Definition: *Synchronization* is the way by which processes that share the same memory space are managed in an operating system. It helps maintain the consistency of data by using variables or hardware, so that only one process can make changes to the shared memory at a time.

- Solutions include **semaphores** and **mutex locks**.

  Mutex is a locking mechanism used to synchronize access to a resource in the critical section (which makes sure only one process is modifying the shared data). Here, we use a LOCK that is set when a process enters from the entry section and gets unset when the process exits from the exit section.

  Semaphores are signaling mechanisms and a process can signal a process that is waiting on a semaphore. This is different from a mutex since a mutex can only be notified by the process that sets the shared lock.

  - Semaphores use the wait() and signal() functions for synchronization among the princess.

Definition: *Deadlock* is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

Disposal Methods

These are functions that the programmer must manually call to free non-memory resources (e.g., network connections).

A Final Word on C and C++ and Safety

C++ is the most memory unsafe language in wide use today!

It is susceptible to memory leaks!

Mutability/Immutability

Immutability: Reduces multithreading bugs

If a value can not change, you can NOT have race conditions.

Writing Garbage Collection Friendly Code

Although the GC is quite adept at automatically reclaiming unreachable memory and avoiding memory leaks, a malicious attacker can nevertheless launch a **denial-of-service (DoS) attack** against the GC, by inducing abnormal heap memory allocation / prolonged object retention.

For example, the GC could need to halt all executing threads to keep up with incoming allocation requests that trigger increased heap management activity. **System throughput** rapidly diminishes in this scenario.

Definition: *A Denial-Of-Service (DoS) Attack* is an attack meant to shut down a machine or network, making it inaccessible to its intended users. DoS attacks accomplish this by flooding the target with traffic, or sending it information that triggers a crash. In both instances, the DoS attack deprives legitimate users (i.e. employees, members, or account holders) of the service or resource they expected.

Definition: *System Throughput* is the total amount of items processed/produced by a system over a defined period of time. An example could be 1,000 items over 100 time units.

Definition: *Latency* is the time it takes for data to pass from one point on a network to another.

OR

Latency is the delay between a user's action and a web application's response to that action, often referred to in networking terms as the total round trip time it takes for a data packet to travel.

Lecture 10: Function Palooza, Part 1

Positional Parameters vs Named Parameters

Positional Parameters: The order of the arguments must match the order of the formal parameters.

Benefits are less wordy syntax, and that we do not need to specify the parameter name and value of reverie parameters.

Named Parameters: The caller explicitly specifies the name of each formal parameter for each argument.

Benefits are that you can add parameters and shift their order around more easily since each parameter is named. A change in order will NOT cause a bug and it overall makes code more readable even if it is more verbose.

Optional Parameters: Using Default Values

In languages like C++ and Python without mandatory argument labels, default parameters must be placed at the end of the parameter list.

Optional Parameters: Using Default Values

Python enables optional parameters without default values.

A function can check if a given argument was present when the function was called and act accordingly.

- You then add a special parameter preceded by two asterisks as a formal parameter to your function.

```
# Python optional parameters
def net_worth(assets, debt, **my_optinals):
        …

print("Net-worth: ", net_worth(10000, 2000))
print("Net-worth: ", net_worth(10000, 2000, inheritance = 50000)
```

- Benefits of default values: Makes code easier to read, makes changes much easier – since we can change a default parameter value in one place rather than 50 places.

"Variadic" Functions

You can pass a variable number of arguments to a function.

The classic example of this is the print function:

```
print(1)
print(1,"a")
print(1,3.14159,"c",4,"foobar")
```

Most languages gather **variadic arguments** and add them to a container (e.g., to an array dictionary or tuple) and pass that container to the function for processing.

A notable exception is C++ which requires use of convoluted library functions to access variadic arguments!

- In python, variadic arguments are identified by a formal parameter with a "*" in front of it.
- Python creates a tuple containing every variadic argument and passes the type to the variadic parameter.

C++ Variadic Example

In C++, variadic arguments are identified by a formal parameter of "...".

C++ does NOT have any way to determine the number of variable arguments, so we have to somehow specify this.

Parameter Passing Semantics

These describe how arguments are passed to functions.

The four most common approaches are closely related to binding semantics.

1) **Value Semantics → Pass by Value**

   Pass by Value: the formal parameter gets a copy of the argument's value/object.

2) **Reference Semantics → Pass by Reference**

   Pass by Reference: the formal parameter gets a copy of the argument's value/object.

3) **Object Reference Semantics → Pass by Object Reference**

   Pass by Object Reference: the formal parameter gets a copy of the argument's value/object

4) **Name Semantics → Pass by Name**

   Pass by Name: the parameter points to an expression graph that represents the argument.

Pass by Value

Each argument is first evaluated to get a value, and a copy of the value is then passed to the function for local use.

> We first evaluate the expression, and send a copy of the result over to the formal parameter.

Lecture 11: Function Palooza, Part 2

Pass by Reference

Now, we secretly pass the address of each argument (whether it be value/object) to the function.

In the called function, all reads/writes to the parameters are directed to the data at the original address.

- All changes to reference parameters modify the original value/object (using the passed in address).

Aliasing occurs when two parameters unknowingly refer to the same value/object and unintentionally modify it.

This can cause subtle and difficult to find bugs.

Pass by Object Reference

All values/objects are passed by pointer to the called function. The called function can use the pointer to read/mutate the pointed-to-argument.

When passing by object reference, we can NOT use assignment to change the value of the original value/object.

- Assignments of object references never change the passed-in value/object.
- They just change where the local object reference points to!
- This is different from pass by reference where assignment *can* change the original value.

Pass by Name/Need

Each parameter is bound to a pointer that points to an expression graph (a **thunk**) which can be used to compute the passed-in argument's value.

A **thunk** is typically implemented as a lambda function that can be called to evaluate the full expression graph and produce a concrete result when it's needed.

- In **pass-by-need**, once an expression graph is evaluated, the computed result is **memoized** (cached) to prevent repeat computation.

Parameter Passing by Language

C++:

Pass by Value
Pass by Reference
Pass by Object Reference (Pointer)
Pass by Macro Expansion

Haskell:

Pass by Need

Python:

Pass by Object Reference (Pointer)

Returning Values and Error Handling

Bugs vs. Errors vs. Results

A Bug: A flaw in a program's logic – the only solution is aborting execution and fixing the bug

Examples:

- Out of bounds array access
- Dereference of a nullptr
- Divide by Zero
- Illegal Cast between types

Unrecoverable Errors: Non-bug errors where recovery is impossible, and the program must shut down

Examples:

- Out of memory errors
- Network host not found
- Disk is full

Recoverable Errors: Non-bug errors where recovery is possible, and the program may continue executing

Examples:

- File not found
- Network service temporarily overload

A Result: An indication of the outcome/status of an operation.

Examples:

- Result of container.find(value)
- Is the password correct

Bug, Error and Result Handling Techniques

The major "handling" paradigms provided by languages:

1) **Roll Your Own** (Older languages: C): The programmer must "roll their own" handling,

like defining enumerated types (success,error) to communicate results.

2) **Error Objects**: Error objects are used to return an explicit error result from a function to its caller, independent of any valid return value.
3) **Optional Objects** (Haskell): An "Optional" object can be used by a function to return a single result that can represent either a valid value or a generic failure condition.
4) **Result Objects** (C++, Haskell): A "Result" object can be used by a function to return a single result that can represent either a valid value or a specific Error Object.
5) **Assertions/Conditions** (Most Languages): An assertion clause checks whether a required condition is true, and immediately exits the program if it is not.
6) **Exceptions and Panics** (C++, C#, Java, Python, …): f() may "throw an exception" which exits f() and all calling functions until the exception is explicitly "caught" and handled by a calling function or the program terminates.

Error Objects

Languages with error objects provide a built-in error class to handle common errors.

Error objects are returned along with a function's result as a separate return value.

You can define **custom** error classes with fields that are specific to your error condition.

Optionals

We can think of an **optional** as a struct that holds two items: a value and a Boolean indicating whether the value is valid.

Since you only have a simple Boolean to indicate success/failure (not a detailed error description), you only want to use Optional if there's an **obvious, single failure mode**.

Result Object

A "Result" Object can be used by a function to return a single result that can represent either a **valid value** or a **distinct error**.

Think of a Result as a struct that holds two items: **a value** and **an Error object** with details about the nature of the error.

- Since a Result contains a **full Error object**, you can use it when there are multiple distinct failure modes that need to be distinguished and handled differently.

Assertions: Pre and Post Conditions and Invariants

An assertion is a statement/clause inserted into a program that verifies assumptions about your

program's state (its variables) that must be true for correct execution.

We typically use assertions to verify…

Preconditions: A precondition is something that must be true at the start of a function for it to work correctly.

Post Conditions: A postcondition is something that the function guarantees is true once it finishes.

Invariants: An invariant is a condition that is expected to be true across a function or class's execution.

An assertion tests a particular condition and terminates the program if it's not met.

An assertion states what you expect to be **true**. Your program aborts if it's **not true**!

We can handle **invariants** with **asserts**.

In a class, an invariant is a particular state that we always expect to be maintained.

- e.g., in a Stack class, the **# used slots** + **# free slots** must always equal the stack's maximum size.
- While it's impossible to maintain an invariant across every line of code, you should maintain invariants at the function level.

Assertions: Analysis

Questions to Consider:

1) When would you use an assertion instead of an error object, optional, or result object?
2) Why would you want a program to crash if an assertion fails, rather than handling it gracefully?

Answers:

An assertion is meant to detect a fundamental flaw or bug in your program – some assumption you're making that's been violated. You wouldn't use an assertion for a recoverable error/status result like you would an error object, result object or optional.

Exception Handling

With **other** error handling approaches, error checking is woven directly into the code, making it harder to understand the core business logic.

With **exception handling**, we separate the handling of exceptional situations/unexpected errors from the core problem-solving logic of our program.

- Thus, errors are **communicated and handled independently** of the mainline logic.

- This allows us to create **more readable code** that focuses on the problem we're trying to solve and isn't littered with extraneous error checks that complicate the code.

There are two participants with exception handling: a **catcher** and a **thrower**.

The catcher has two parts:

1) A block of code that **tries** to complete one or more operations that might result in an unexpected error.
2) An **exception handler** that runs iff an error occurs during the tried operations, and deals with the error

The thrower is a function that performs an operation that might result in an error. If an error occurs, the thrower creates an exception object containing details about the error, and "throws" it to the exception handler to deal with.

- If any operation performed in the **try block** results in a thrown exception, the exception is immediately passed to the exception handler for processing.

<u>What is An Exception?</u>

An exception is an object with one or more fields which describe an **exceptional situation**.

At a minimum, every exception has a way to get a description of the problem that occurred, e.g., "division by zero."

- But programmers can also use **subclassing** to create their own exception classes and encode other relevant info.

Lecture 12: Function Palooza, Part 3

Exception Handling

An exception may be thrown an arbitrary number of levels down from the catcher…

And will automatically terminate every intervening function on its way back to the handler!

- An exception handler can specify exactly what **type of exception(s)** it handles.
- A thrown exception will be directed to the closest handler (in the call stack) that covers its exception type.

If an exception is thrown for which there is no compatible handler, the program will just terminate.

- This is the equivalent of a "panic" which basically terminates the program when an unhandle-able error occurs.

Different types of exceptions are derived from more basic types of exceptions.

- So if you want to create a catch-all handler, you can have it specify a (more) basic exception class.
- That handler will deal with the base exception type and all of its subclassed exceptions.

Instead of using the **catch** keyword like C++, Python uses the **expect** keyword to catch various exception types.

- Also, Python uses **raise** instead of **throw** to throw an exception.
- Python can also support nested exceptions using the inner handlers and outer handlers.

Exception Handling Guarantees

The "No-throw" Guarantee (aka Failure Transparency) – More Preferred

A function guarantees it will not throw an exception. If an exception occurs in/below the function, it will handle it internally, and not throw.

- For example, this is required of **destructors, functions that deallocate memory, swapping functions**, etc.

Strong Exception Guarantee – Kind Of Preferred

If a function throws an exception, it guarantees that the program's state will be "rolled-back" to the state just before the function call.

- For example, vec.push_back() ensures the vector doesn't change if the

push_back() operation fails.

Basic Exception Guarantee – Less Preferred

If a function throws an exception, it leaves the program in a valid state (no resources are leaked, and all invariants are intact).

- State may not be rolled back, but at least the program can continue running

Panics

Like an exception, a **panic** is used to **abort execution** due to an exceptional situation which cannot be recovered from (e.g., an unrecoverable error).

A panic immediately terminates the program and can't be caught like an exception!

Essentially, you can think of a panic as an exception which is never caught, and thus which causes the program to terminate.

Panics contain both an **error message** and a **stack trace** to provide the programmer with context as to why the software failed.

Exception Handling and Panics: Analysis

With exception handling, is there any chance an error will be ignored if the programmer forgets to add a try/catch?

Chance an error will be ignored? No way. If the programmer ignores an exception, the program will crash and everyone will know there was an error. So this forces proper error handling.

With exception handling, what things must a programmer consider that weren't relevant for the other error handling techniques?

Considerations: Exceptions could result in memory/resource leaks and will impact the execution path/control flow in sometimes unexpected ways.

Error Handling Best Practices

Use assertions

- To check for errors that should never occur if your code is correct (e.g., bugs, unmet preconditions, or violated invariants)
- To build **unit tests** that validate individual classes/functions.

Definition: *Unit Tests* are typically automated tests written and run by software developers to ensure that a section of an application (known as the "unit") meets its design and behaves as intended. In procedural programming, a unit could be an entire module, but it is more commonly

an individual function or procedure. In object-oriented programming, a unit is often an entire interface, such as a class, or an individual method. By writing tests first for the smallest testable units, then the compound behaviors between those, one can build up comprehensive tests for complex applications.

Use exceptions

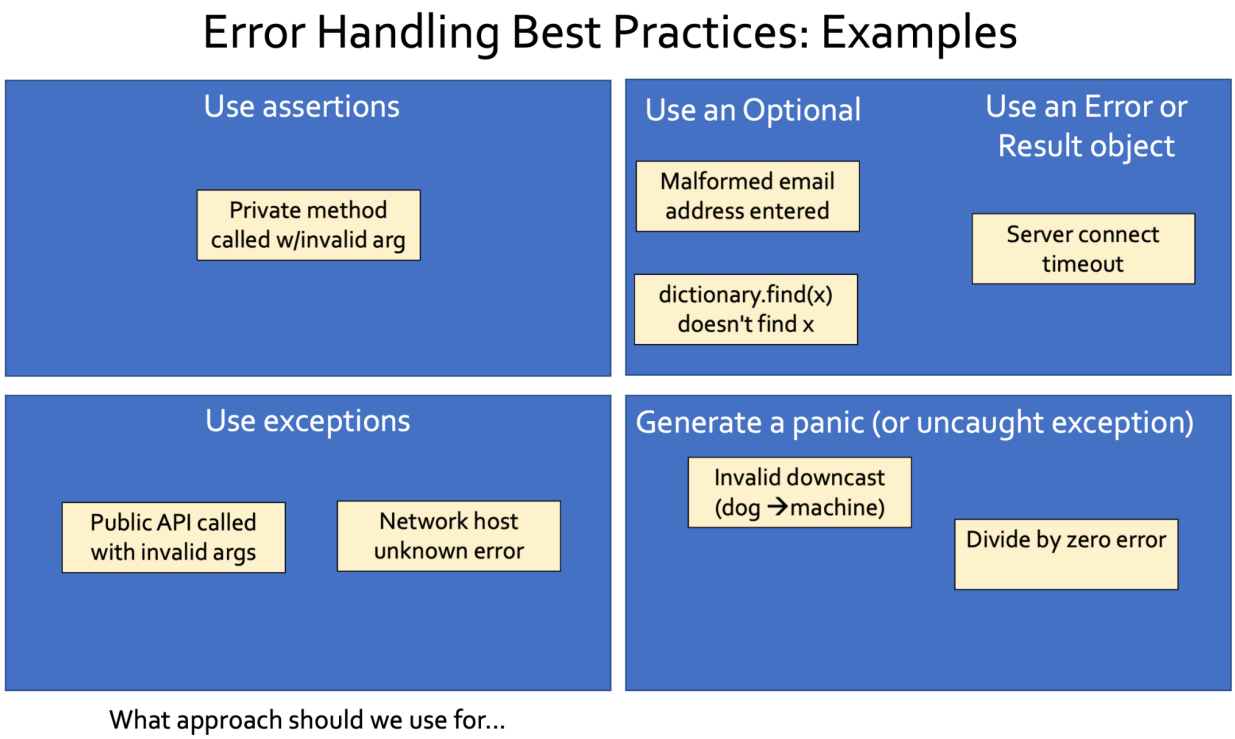
When a function is unable to fulfill its "contract," AND it's an error that occurs < 1% of the time, AND the failure can be recovered from

Generate a panic (or uncaught exception)

When there is no reasonable way to recover from an error

Use an Optional/Use an Error or Result object

When there's only one possible failure mode/When there are multiple failure modes and you need to pass the details of the error to the caller



First Class Functions

Big Picture

Functions are called **first-class citizens** because they are data objects and can be manipulated like any other data in a program.

In languages with **First Class Functions**, functions can be passed/returned to/from other functions, stored in data structures, compared for equality and expressed as anonymous, literal values**.** Variables can also be assigned to functions.

In languages with **Second Class Functions**, functions can be **passed as arguments** but not returned or assigned to variables.

In languages with **Third Class Functions**, functions can be **called** but nothing else!

First-class Functions in Different Languages: C++

In C++, first class functions are implemented with function pointers:

The general syntax to define a function pointer is:

returntype (*)(type1 param1, type2 param2, …)

C++ functions can return pointers to functions.

A function that accepts a function as a parameter or returns a function is called a **higher-order function**.

Anonymous (Lambda) Functions

Big Picture

A **lambda function** is a function that does not have a function name – it's anonymous:

Typically, we pass a lambda function as an **argument** to another function or **store** it in a variable.

- Lambdas are used when a **short, temporary** function is needed just once in a program, and it doesn;t make sense to define a general-purpose named function.

Lambda Function Details – with C++

Every C++ Lambda function has three different parts:

1) **Free Variables to Capture** – What variables defined outside the lambda function that should be available for use in the lambda function when it is later called?
2) **Parameters & Return Type** – What parameters does the lambda function take and what type of data does it return.
3) **Function Body** – The body of the lambda function that performs its operations.

A “closure” contains **captured variables** and the **lambda function** itself.

Lambda Function Details – Capture by Value in Java

Java lambda functions capture free variables by value, just like C++!

And as we know, Java uses object references.

- Capture-by-value and object references work together.
- Languages that **capture by value** and use object references capture only the **object reference’s** value and not the full pointed-to object in their closure!
- Only the object reference value is captured, not the referenced object.

So changes to a captured object impact the **original object** outside the closure, not some cloned copy inside the closure!

Lambda Function Details – C++ Capture by Reference

C++ can also capture free variables by reference.

If you place an & in **front of a captured variable**, C++ will make it a "capture by reference.”

- The captured variable will be a reference to the original variable which allows you to change a value outside the lambda function!
- If our captured variable is a reference to the original variable, then it enables the lambda to change variables outside its closure.
- BE CAREFUL! If your lambda function refers to an out-of-scope variable, bad things will happen!

Lambda Function in C++ Capture Summary

There are three variable capture approaches:

1) **Capture by Value** – A copy of the captured variable’s value is frozen and added to the closure.
2) **Capture by Reference** – A reference to the captured variable value is added to the closure.
3) **Capture by Environment** – An object reference to the lexical environment where the lambda was created is added to the closure.

Lambda Function Details

Python supports **capture by environment** semantics.

```
# Python capture by environment example
def foo():
        Q = 5
f = lambda x: print("q*x is: ", q*x)
f(10)
```

When you define your lambda, it creates a **closure** containing:

1) Your **lambda function**
2) An object reference to the current **lexical environment**.

The lexical environment is a data structure that holds a mapping between every in-scope variable and its value.

- That includes all variables in the current activation record (locals, statics), and all global variables.
- When running the lambda, it looks up each free variable in the lexical environment to obtain its value.
- Also, when the lambda runs, it always looks up and uses the latest value of the variable in its own lexical environment.

Polymorphism

Big Picture

Polymorphism is a technique where we define a function or class that is able to operate on multiple different types of values.

Our goal is to express algorithms with minimal assumptions about the types of data they operate on.

Types of Polymorphism in Statically Typed Languages

There are three primary types of polymorphism in statically typed languages.

Subtype polymorphism
Ad-hoc
Parametric

Ad-hoc Polymorphism

We define **specialized versions** of a function for **each type** of object we wish it to support.

The language decides which version of the function to call based on the **types of the**

arguments.

- This is used all the time in C++ for operator overloading.
- We can NOT have ad-hoc polymorphism in dynamically typed languages like Python because there's no way to define multiple versions of a function with different parameter types.

Parametric Polymorphism

We define a **single**, parameterized version of a class or a function that can operate on **many** potentially **unrelated** types.

There are two different types:

Templating
Generics

Parametric Polymorphism: Templates

In a language like C++ that uses the **template approach**, each time you use the template with a **different type**, the compiler generates a **concrete version** of the template function/class by substituting in the **type parameter**.

Then it just compiles the newly generated functions/classes as if they were never templated in the first place!

- Any operations you use in your templated code must be supported by the types being templated.
- This is like duck typing in Python, but is NOT necessarily duck typing.

Templates: Analysis

Templated code is the same in terms of efficiency (of runtime) as an equivalent function that does NOT use templates, but it is slower at compile time!

Compiler errors in templated code are more cryptic than errors in other types of code because the source code was generated by the compiler and you get errors from the code that you never wrote.

- Also, If you make a change to a templated class, you would need to recompile all source files that use the template. If we had 1000 source files that used a template and we changed the template, we would have to regenerate/recompile each of those 1000 files to change the classes/functions associated with the template.

C Macros – Sort of like Templates

There are multiple ways of implementing template-like parametric polymorphism.

With C macros, the compiler preprocessor would basically do **search** and **replace**.

- The compiler does NOT generate a new function for each parameterized type.

Parametric Polymorphism: Generics

Each generic class/function is **compiled on its own**, separate from any code that later uses it.

Because of this, a simple generic's code can NOT make any assumptions about what types it might be used with.

When you later compile the code that uses the generic, the compiler will ensure the code uses the generic's interfaces completely.

If we want, we can place restrictions on what types can be used with our generic. This is called **bounding.**

- For example, if we wanted to limit our generic to working with objects that behave like a duck, we can!
- Then we can update our generic to use features consistent with this interface and now we can only use our generic with classes that support this interface.

Why Do We Need Parametric Polymorphism?

In languages like Java and C#, every class is implicitly derived from a built-in **Object** class.

If that's the case, why don't we skip all the complicated syntax and just use inheritance-based (aka subtype) polymorphism for our generics?

- Then, we would get none of the type-safety that generics provide us with.

Generics: Analysis

When you use a generic class/function with a new type, the code is guaranteed to be type-safe!

If you make a modification to code in a generic function/class (but its interface doesn't change), we do NOT have to recompile all of your other source code that uses that generic because as long as the generic maintains the same interfaces (public methods, types), it can just be used by other source files.

Errors are more meaningful for generics because the errors will be generated for either the generic itself of the code that calls it, not for some compiler-generated code (like templates).

Downsides to generics are that it still has complicated syntax and that it does NOT allow for duck typing like templates unless you use a bounded type.

Lecture 13: Object Oriented Programming (OOP), Part 1

What is OOP?

It is a programming paradigm based on the concept of objects.

We use objects to represent self-contained entities like ADTs (Abstract Data Types).

OOP History

In 1991, Sun Microsystems began creating a new OOP language called **Java**, designed for consumer devices.

Rather than compile Java source to machine code, the team decided to compile Java to "**bytecode**."

- Bytecode is like a machine language, but not targeted at a particular microprocessor.

The big idea was to create an **interpreter** (called the Java Virtual Machine) for this bytecode format and **port it** to lots of different devices...

- To run the same UI and logic on dozens of different devices independent of their CPU.

Microsoft created C# which is also used for Unity – AR and VR applications!

The Essential Components of OOP

Encapsulation
Classes
Interfaces
Objects
Inheritance
Subtype Polymorphism
Dynamic Dispatch

Encapsulation

This is the guiding principle behind the OOP Paradigm

There are two facets:

1) We bundle related public interface, data, and code together into a cohesive, problem-solving machine
2) We hide data/implementation details of that machine from its clients – forcing them to use its public interface.

Benefits of Encapsulation

Simpler Programs

We have reduced coupling between modules, simplifying our programs

- Since each class only depends on the public interface of other classes, it prevents deep coupling between components, which makes code less complex and less buggy.

Easier Improvements

We can improve implementations without impacting other components.

- Since each class only depends on the public interface of other classes, we can drastically change how a class works internally without breaking other classes that use it.

Better Modularity

We can build a class once and use it over and over in different contexts.

- We can design, implement, and test a class largely in isolation of other code.Once completed, it can safely be reused as-is by other code via its public interface.

Classes

This is a blueprint that specifies data fields and methods that can be used to create objects.

Class Components:

1) Class Name
2) Class Public Interface

A Class Definition Implicitly Defines a Type

When you define a new **class**, it automatically creates a new **type** of the same name.

Lecture 14: Object Oriented Programming (OOP), Part 2

Interfaces

An **interface** is a **related group of function prototypes** that describes behaviors that we want one or more classes to implement.

```
// C++ interface definition for shapes
class Shape {
 public:
   virtual double area() = 0;
   virtual double perimeter() = 0;
   virtual void scale(double factor) = 0;
};
```

- Notice, there's no actual { code }, just a prototype that describes each operation and its inputs/outputs.
- These are called **pure virtual** functions – they're basically function prototypes.
- Overall, an interface specifies **what** we want one or more classes to do, but **not how**.

Interfaces: Why?

If we want a bunch of classes to provide a common set of behaviors but there's no common implementation to inherit, we can define an interface!

Each Interface Defines a New Type

Just as with **classes**, when you define a new **interface**, it automatically creates a new **type** of the same name!

If each interface defines a type, and each class defines a type, and a class implements an interface – does the class have two types?

- Yes! A class that implements one or more interfaces is a subtype of each interface's type, and so it has its own type, and is also a subtype of each of the interface(s).

Objects

An object is a distinct value, often created from a class blueprint – each object has its own copy of fields and methods.

Each object has its own distinct member variables and its own logical "copies" of all the methods, which operate on the object's members.

- Another name for this bundle of values and methods is an "instance.

Objects… But Not Necessarily Classes

Not all OOP languages use classes!

For example, JavaScript has only objects – not classes!

Classes, Interfaces, and Objects

Why even support the notion of a class in a language? What are the pros and cons of using a model like JavaScript with just objects?

- Classes enable us to create a consistent set of objects, all generated in a consistent way based on the class specification
- Classes enable us to define a new type, which we'll see is useful for polymorphism in statically typed languages

JavaScript can dynamically add methods to objects, is there any reason we can't dynamically add/remove methods to/from classes?

- Classes enable us to create a consistent set of objects, all generated in a consistent way based on the class specification
- While most/all static languages don't allow this, dynamically typed languages absolutely allow this, Python being one example. In a statically typed language, adding a new method would potentially change the class interface which would complicate compilation/interpretation, but technically this could be done too.

Classes

Class Fields and Class Methods

A **class field** is one that's associated with the overall class – it's stored once in memory and is usable by all of the class's objects.

A **class method** is similarly associated with the overall class, and operates on the field. It does not have access to the objects!

Class Fields

If we want to define a "singleton" field that's shared amongst all objects, this is a **class field.**

Instance fields are when each object/instance has its own distinct copy of the field.

Uses:

Defining Class-Level Constants
Counting the Number of Objects
Assigning Each Object a Unique ID

Class Methods

If we have a method that never needs to access any of a class's instance variables, then we can make our method a **class method**.

In C++, the static keyword defines a method as being a class-level method.
A class method is one that is shared across all objects, and thus may only access class-level variables.

Class Fields and Methods: Python

```
# defining a class method
@classmethod
def function(class_arg):
    return …
```

class_arg refers to the class argument that refers to the class

This and Self

These can be an object reference (like "self" is in Swift or Python) or a pointer (like "this" is in C++).

In python, the language is more explicit about the object pointer/reference.

- Every instance method must explicitly define self as its first parameter.

Class Access Modifiers

There are three levels of access visibility in most languages.

1) **Public** – A public method or data member in class C may be accessed by any part of your program.
2) **Protected** – A protected method or data member in class C may be accessed by C or any subclass derived from C.
3) **Private** – A private method or data member in class C may only be accessed by methods in class C.

public, protected and private generally have the same semantics across languages

- If you omit an explicit access modifier, different languages have different defaults: some default fields/methods to private, some default to public, etc.

Access Modifiers in Python

Python supports public, protected and private concepts, but does not have explicit keywords for the access modifiers.

Encapsulation Best Practices

1) Design your classes such that they hide all implementation details from other classes.
2) Make all member fields, constants, and helper methods private.
3) Avoid providing getters/setters for fields that are specific to your current implementation, to reduce coupling with other classes.
4) Make sure your constructors completely initialize objects, so users don't have to call other methods to create a fully-valid object.

Properties Accessors, and Mutators

While our goal in OOP is **encapsulation**, sometimes we want to expose a field (aka property) of an object for external use.

To do so, we create methods called **accessors (aka getters)** and **mutators (aka setters)** to get/change the value of a field.

- Accessors and mutators for a **property** enable us to hide its implementation and more easily refactor.
- In fact, they even allow us to create the illusion that a class holds a field, when there's no actual **member variable**.

Inheritance

This is a programming technique where we define a new class by basing it on an existing class or interface.

It lets us reuse code and to ensure that different classes behave in a standardized manner.

What is Inheritance?

There are four types of inheritance:

Interface Inheritance
Implementation Inheritance:
Subclassing/Hybrid Inheritance
Prototypal Inheritance

Interface Inheritance

Step 1: you define a public interface for a set of related methods
Step 2: A derived class inherits the interface and provides implementations for its methods

- If a class provides implementations for all the functions in the interface, the class is said to **inherit, implement, or support the interface**.

Supporting Interfaces

With interface inheritance, we can have totally unrelated classes support the same interface.

Lecture 15: Object Oriented Programming (OOP), Part 3

Supporting Interfaces

With interface inheritance, we can have totally unrelated classes support the same interface.

Also, a given class may support multiple interfaces.

Interface Inheritance Use Cases

A use case would be for callbacks. You can have a class implement a callback interface, and then objects can receive callback notifications when an event occurs (e.g., a network connection has terminated).

Definition: A callback or callback function is any reference to executable code that is passed as an argument to another piece of code; that code is expected to call back the callback function as part of its job.

Another use case would be for enabling interaction over container elements!

When to Use Interface Inheritance: Best Practices

When you have a **can-support** relationship between a class and a group of behaviors

"The Car class can support washing."

When you have different classes that all need to support related behaviors, but are NOT related to the same based class.

"Cats and Dogs can both be washed, but aren't related by a common base."

Interface Inheritance: Pros and Cons

Pros:

You can write functions focused on an interface and have it apply to many different cases.

A single class can implement multiple interfaces and play different roles.

Cons:

Interface Inheritance doesn't facilitate code reuse.

Subclassing – aka Hybrid Inheritance

This is the inheritance most people are used to.

The base class defines **public** methods with optional implementations, as well as

private/protected methods.

Our base class defines a set of public methods, which implicitly define the class' public interface.

The derived class inherits the base class' public methods – and thus, it inherits its interface.

- It also inherits the implementations of the protected methods.

The derived class inherits the base class's **public interface** and all of its methods' implementations.

- The derived class may add **new methods** and override existing implementations.
- In C++, only virtual methods should be overridden

Subclassing in Python

This is how we implement an abstract method in Python: using **pass**

```
def area(self);
    pass
```

A leading underscore is used to indicate a protected method:

```
def _area(self);
    pass
```

In python, each derived object has multiple dictionaries (one pre class) to store members and methods. Using a member/method initiates a search through each dictionary.

When to Use Subclassing: Best Practices

1) When there's an is-a relationship:
   "A Circle is a type of Shape"
   "A Car is a type of Vehicle"
2) When you expect your subclass to share the entire public interface of the superclass AND maintain the semantics of the super's methods
3) When you can factor out common implementations from subclasses into a superclass:
   "All Shapes (Circles, Squares) share x,y coordinates and a color."

When NOT to Use Subclassing

Reason #1: Our derived class doesn't support the full interface of the base class.
Reason #2: The derived class doesn't share the same semantics as the base class.

Subclassing Alternative: Delegation

With delegation, a class embeds the original objects as a member, and calls its functions directly.

We simply ask the method from the contained object to do the work.

Use delegation when you want to leverage another class's implementation but not support its interface.

Subclassing Pros and Cons

Pros:

Eliminates code duplication/facilitates code reuse

Simpler maintenance – fix a bug once and it affects all subclasses

Cons:

Often results in poor encapsulation (derives class uses base class details)

Any change to superclasses can break subclasses

Overall, a change to a base class unknowingly breaks a derived class!

Implementation Inheritance (Not Popular)

The derived class inherits the **method implementations** of the base class, but NOT its **public interface**.

Our derived class privately inherits from the base class.

- This causes the derived class to inherit the implementation of the base class, but hide the base's public interface
- Our code can use the derived class's public interface.

When to Use Implementation Inheritance?

This is strongly frowned upon!

Instead, it's recommended to use **composition** and **delegation**.

- **Composition** would be defining a member variable inside.
- **Delegating** is delegating by forwarding or calling.

We NEVER want to use implementation inheritance over composition and delegation.

One More Type of Inheritance: Prototypal Inheritance

We use Prototypal Inheritance in a language like Javascript that has **objects** without **classes**.

Every Javascript object has a hidden reference to a parent (protype) object.

By default, the parent is Javascript's built-in empty object.

- But… you can set an **object's prototype reference** to point at another object!
- If we redefine the value of a field found in the prototype, this shadows (hides) the original value.

Inheritance: Construction

Construction follows the same basic pattern in all languages.

When you **instantiate** an object of a derived class, this calls the derived class' **constructor**.

Every derived class **constructor** does two things:

1) It calls its superclass's constructor to initialize the base part of the object
2) It initializes its own parts of the object

Every derived class constructor follows this model..

- Finally the base class constructor just initializes the base part of the object.

In most languages, if a **base class constructo**r takes no parameters…

Then your derived constructor need not explicitly call it – the language will call it automatically!

- A notable exception is Python, which always requires an explicit constructor call.

Inheritance: Destruction and Finalization

In C++, **destruction** starts with the most derived subclass and ends with the base class.

Each destructor runs its { code } and then implicitly calls the destructor of its superclass.

But what about **finalizers** – how do they work with inherited classes?

It depends! C++ destructors do this **implicitly**.

Check your language spec for details! And remember, finalizers often don't run at all - so try to avoid them!

Inheritance: Overriding Method and Classes

Overriding is when a derived class defines its own **implementation** for a method initially

defined in one of its superclasses.

It enables a subclass to specialize its behavior, diverging from the behavior of a superclass.

- Overriding is a distinct concept from **overloading**, which is about defining multiple versions of a function which each take different parameters.

Controlling Method Overriding

When designing a base class, you must designate which methods can be **safely** overridden and methods that **shouldn't** be overridden.

In some languages, you declare explicitly which methods MAY be overridden.

- C++ has an "override" keyword which is optional, but good style. It indicates that a method is overriding the base implementation.
- A method without "virtual" indicates it must not have its implementation overridden.

Ensuring Proper Overriding

As we've seen, many OOP languages offer a way to indicate that a derived class is explicitly overriding a method from a base class.

We need a keyword to indicate that a derived method overrides a base method

Controlling Class Overriding

In some languages, you can also designate that an entire class must NOT be inherited from!

In Python, you can apply the **final decorator** to a class.

- Good OOP style dictates that you should be rigorous about indicating finality – it helps to reduce bugs and increase code readability.

Accessing a Superclass Version of an Overridden Method

Sometimes a derived version of a method needs to use a superclass implementation to get things done.

In C++, a method may call a superclass version of a method by prefixing with the appropriate **class name**.

In Python, you may use a **class name** OR a **reference** to super() to call a superclass method! (Just don't forget to pass the self object reference in!)

There are two approaches: using a superclass keyword (e.g., super, base) or using the superclass's name.

Lecture 16: Object Oriented Programming (OOP), Part 4

Multiple Inheritance and Its Issues

Multiple inheritance is when a derived class inherits implementations from two or more superclasses.

It is considered **forbidden**!

- In cases where you want to use multiple inheritance, have your class implement multiple interfaces instead.

A class can be subclasses derived from multiple base classes in some languages like C++

Multiple Inheritance: The Diamond Pattern

This kind of problem can happen any time you inherit **method implementations**!

Multiple Inheritance

A simple example of an issue in multiple implementation inheritance even when the diamond pattern is not used:

If base classes A and B both define method f() { ... } , even if A and B are not derived from the same base class, and derived class D inherits from both A and B, then a call to method f() will be an ambiguous call, unless D redefines F! Should a D.f() go to A.f() or B.f()?

A language enable the programmer to eliminate this issue by:

Allowing the programmer to explicitly specify which version of f() to use when using a D object, e.g. obj.A.f() or obj.B.f(), eliminating any ambiguity.

Types of Inheritance

Single Inheritance: When a subclass inherits only from one base class

Multiple Inheritance: When a subclass inherits from multiple base classes.

Hierarchical Inheritance: When many subclasses inherit from a single base class

Multilevel Inheritance: When a subclass inherits from a class that itself inherits from another class. The transitive nature of inheritance is reflected by this form of inheritance.

Hybrid Inheritance: A combination of 2 or more forms of inheritance (when a subclass inherits from multiple base classes and all of its base classes inherit from a single base class).

Abstract Methods and Classes

Methods that define an interface, but don't have an implementation.

They define "**what**" a method is supposed to do and its inputs and outputs – but not **how**.

When defining a base class, most languages allow the programmer to specify one of more **abstract methods**, which are just function prototypes without implementations.

```
class Shape { // C++ class
public:
  // area and rotate are "abstract" methods
  virtual double area() const = 0;
  virtual void rotate(double angle) = 0;
  virtual string myName()
    { return "abstract shape"; }
};
```

This class holding the abstract method is called an **abstract base class** because it lacks implementations for one or more functions.

We can NOT instantiate an object of A.B.C. like Shape, however:

```
int main() {
  Shape s;  // ?????
  cout << s.area(); // ????
}
```

An **abstract method** forces the programmer to redefine the method in a derived class, ensuring they don't accidentally use a dummy base implementation.

Each Abstract Class Defines a New Type

Just as with **concrete classes**, when you define a new **abstract class**, it automatically creates a

new type of the same name.

We can NOT instantiate new objects with an abstract type since its class is not fully implemented.

When To Use Abstract Methods and Classes: Best Practices

We use them in these circumstances:

- Prefer an **abstract method** when there's no valid default implementation for the method in your base class
- Prefer an **abstract class** over an interface when all subclasses share a **common implementation** (one or more concrete methods or fields)

Abstract Methods and Classes

A private method can NOT be declared as abstract because there's no way to inherit a private method because it is private!

Classes, Interfaces, and Typing

When you define a class or an interface, it implicitly defines a new type.

The type associated with a **concrete class** is specifically called a **value type**.

- Value types can be used to define references, object references, pointers, and instantiate objects.

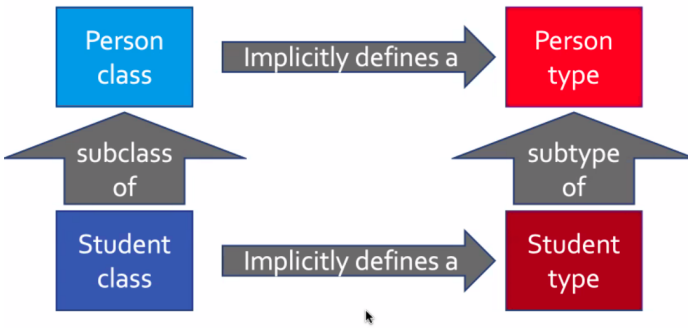
Subclassing (Hybrid Inheritance) and Subtyping

When you define a class or an interface, it implicitly defines a new type.

A class is a class and a type is a type: the two are related.

Types are for type checking.

Classes are for implementation and the code that runs.



Subtype Polymorphism

The ability to substitute an object of a subtype (Circle) anywhere a supertype (Shape) is expected

This is where code designed to operate on an object of **type T** operates on an object that is a **subtype of T**.

- If a function can operate on a **supertype**, it can also operate on a **subtype**.
- The same holds true for supertypes and subtypes associated with inherited classes and for classes that implement interfaces.

We can substitute an object of a subtype anywhere a supertype is expected…

This works because the subclass inherits/implements the same interface as its superclass, BUT…

- We also expect the methods of the subclass to have the same semantics as those in superclass
- If a subclass adheres to the interface *and* semantics of the base class, we can truly substitute any subtype object for a supertype, and our code will still work.

Subtype Polymorphism in Dynamically Typed Languages

There is none! You can have inheritance and duck typing, but NOT subtype polymorphism.

We do NOT have subtype polymorphism where we are accessing a derived object through a base.

Dynamic Dispatch

Imagine we have a variable var that refers to some object:

With Dynamic Dispatch, when you call a method on var, e.g.: var.area()

The actual method that gets called is determined at runtime based on the **target object** that var refers to.

- The determination is made in one of two ways:

1) Based on the target object's class, or
2) By seeing if the target object has a **matching method** (regardless of the object's type), and calling it if one was found.

Dynamic Dispatch in Statically Typed Languages

In statically typed languages, the language examines the class of an object at the time of a method call and uses this to "dynamically dispatch" to the class's method.

Dynamic Dispatch: How Does it Work?

If you were implementing dynamic dispatch yourself, what hidden member variable(s) could you add to all objects to enable your code to find the right method to call?

- The compiler could add a hidden object type member variable to every object which would contain the object's specific type (HybridCar vs Car vs ElectricCar). It could consult this variable to determine a specific object's actual type, and then it's easy to determine which method to call.
- The compiler could also use a **VTABLE** – a pointer table of functions held by each object – to determine which function to call. We'll see that in a second.

When you define a class…

The language creates a table that maps each virtual method to the proper implementation. Each class/type has its own distinct **vtable**!

- It contains an entry for every *virtual/overridable* function in our class.
- For every class with virtual methods, we have a vtable!
- Vtables are created at compile time.

Static Dispatch for Non-Virtual Methods

For these methods, there's no ambiguity of which method to call – there's only one version!

This is called **static dispatch** – the right function can be determined statically at *compile time!*

Dynamic Dispatch in Dynamically Typed Languages

While a class may define a default set of methods for its objects…

The programmer can **add/remove** methods at **runtime** to classes or sometimes even individual objects!

- So when a method is called, the language can **NOT** necessarily rely upon an

object's class to figure out what method to use!

So, how might a language that allows you to customize methods for individual objects determine which method to call?

Answer: The language stores a unique vtable in every object!

- At runtime

Dynamic Dispatch in Dynamically Typed Languages

What is the difference between subtype polymorphism and dynamic dispatch?

- **Subtype polymorphism** is a statically-typed language concept that allows you to **substitute/use** a subtype object (e.g., HighPitchedGeek) anywhere code expects a supertype object (e.g., a Geek). Since the subtype class shares the same public interfaces as the supertype, the compiler knows that the types are compatible and substitution is allowed.
- **Dynamic dispatch** is all about **determining** where to find the right method to call at run-time for virtual/overridable methods. Dynamic dispatch occurs whether or not we have subclassing, as we've seen in dynamically-typed languages like Ruby and Python.

Is dynamic dispatch slower, faster, or the same speed as static dispatch? Why?

Dynamic dispatch is **slower** than static dispatch, since we need to look up function pointers in the vtable at runtime before performing the call.

- Static dispatch: hard-wired @ compile time, much faster: direct machine lang call.

Object Oriented Programming (OOP) Design Patterns

Objects in a Chain of Responsibility must have a common type, but usually they don't share a common implementation. In the Composite pattern, Component defines a common interface, but Composite often defines a common implementation. Command, Observed State, and Strategy are often implemented with abstract classes that are pure interfaces.

(OOP) Design Patterns – Singleton

Normally, we instantiate multiple objects of a given class (e.g., multiple enemy objects of an Enemy class).

But in some cases, we want to have only a single instance/object of a class. Why?

For example, what if it's expensive to establish multiple **network** connections to a database server?

- In this case, we'd want to have a singleton Connection object that manages a

single **connection** to the **Database**.

- As different parts of our program need to use the **Database**, they can ask the"singleton" object to connect on their behalf.

The Singleton Pattern is used in these situations and involves a single class which is responsible for creating an object while making sure that only a single object gets created (we might want to have a single object that connects to a database, shared by multiple **threads** in our program).

Every Singleton class has a **class variable** (not a per-instance variable) that will point at the singleton object (**static variable** = class variable, in C++).

- In a Singleton Class, we make the constructor private – this prevents users from calling it to create **multiple** objects.
- We expose a **get_instance()** class method to get access to the singleton instance/object.

OOP Best Practices: **SOLID**

Single Responsibility Principle
Open/Closed principle
Liskov's Substitution Principle
Interface Segregation Principle
Dependency Inversion Principle

Single Responsibility Principle

Each class should have a single, limited responsibility – if it has more, it should be split into two or more classes.

This makes your classes easier to explain, understand and implement.

Open/Closed Principle

When a class A operates on other classes B, C, D, you should design A such that it can operate on new related classes E, F, G without modification.

This makes your code extensible without needing to be reimplemented!

- We want our base class "**open**" for subclassing and our original code that operates on objects "**closed**" for modification.

Liskov's Substitution Principle

A properly written subclass should be substitutable for its superclass and your code should still

work.

> "Subtype Requirement: Let φ(x) be a property provable about objects x of type T. Then φ(y) should be true for objects y of type S where S is a subtype of T."
>
> - Liskov and Wing 1994

For this to be true, a derived class must support the same **interface** AND the same **semantics** of the base class.

This enables code re-usability – it's also a litmus test for whether subclassing is warranted.

Interface Segregation Principle

No code should be forced to depend on an interface (or base class) that contains methods it does not use.

We can use this to design more granular interfaces!

Then, we can have our concrete classes implement just what they wish to support!

- Overall, this improves code maintainability and prevents hacks for non-implemented methods.

Dependency Inversion Principle

If class A uses class B, then design A to operate on an interface **I**, and have B implement **I**. Don't have A directly use class B.

First Aspect:

- Higher-level classes should not directly refer to/use lower-level classes.
- Instead, lower-level classes should implement an interface... and higher-level classes should be built on that interface.

Second Aspect

- Abstractions (interfaces) should not be designed based on specific details of an implementation.
- **Instead**, implementations should be designed to meet the requirements of the interface.

Overall, this reduces coupling between classes providing flexibility for mixing/matching classes.

Lecture 17: Control Palooza, Part 1: Evaluation and Short Circuiting

Expression Evaluation

Some languages evaluate from right to left, left to right, or do not have a specified order.

Expressions

An expression is a combination of **values**, **function calls**, and **operations** that, when evaluated, produces output values.

An expression may or may not have side effects.

Expressions, Associativity, and Order of Evaluation

```
int c = 1, d = 2;

int f() {
  ++c;
  return 0;
}

int main() {
  cout << c - d + f() ;
}
```

This C++ program will print either -1 or 0.

C++ does NOT specify the order the terms of an arithmetic expression must be evaluated.

```
int c = 0;

int f() {
  ++c;
  return c;
}

void h(int a, int b)
  { cout << a << " "<< b; }

int main() {
  h(c, f());
}
```

In the C++ program above, it is also evident that it does NOT specify what order parameters must be evaluated.

Some languages mandate a left to right evaluation order (C#, Javascript, Java) while others are done (Rust, OCaml).

- We would NOT want a language to specify order if we wanted **compiler**

optimizations.

- The compiler could choose which way to evaluate something. For example, we could run two different choices in **parallel** to optimize whatever the compiler ends up doing.

Regardless of what order each of the terms are evaluated, most languages use left to right associativity when evaluating mathematical expressions.

Short Circuiting

In any boolean expression with AND / OR, most languages will evaluate its sub-expressions from left-to-right…

The moment the language finds a condition that satisfies or invalidates the boolean expression, it will skip evaluating the rest of the sub-expression.

- This can dramatically speed up or code.
- Short circuiting applies to any **boolean expression,** not just if statements.

  Fun Fact: Short Circuiting is AKA Mccarthy Evaluation (from John Mccarthy, who invented Lisp and Garbage Collection).
- Some languages let you choose if you want to use short-circuiting (like Kotlin). Kotlin has two types of boolean expressions: for example, you can use "||" or the "or".

Looping/Iteration

Iteration (Looping)

Three Types:

Counter-Controlled
Condition-Controlled
Collection-Controlled

- Both collection and counter-controlled rely upon iterators under the hood.

Collection-Controlled Iteration

This is when a variety of languages iterate over items in a collection

How Does Iteration Actually Work?

How does iteration actually work under the hood?

Loops are implemented using **iterators** or using **first-class functions**.

Iterators

This is an object that enables enumeration of a sequence of values.

It provides a way to access the values of a sequence without exposing the sequence's underlying implementation.

It can be used to enumerate the values held in a container.

It can be used to enumerate the contents of external sources like data files.

It can be used to enumerate the values of abstract sequences.

- Abstract Sequences: This is where values are not stored anywhere, but generated as they're retrieved via the iterator.

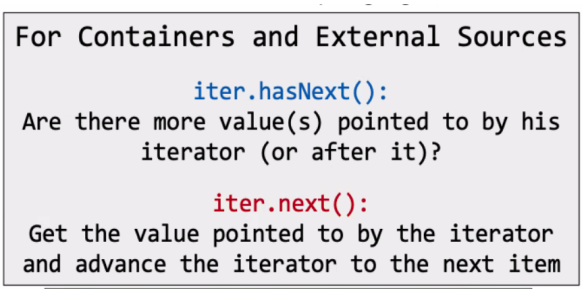
Iterators used with containers and external sources are typically distinct objects from the sequences they refer to.

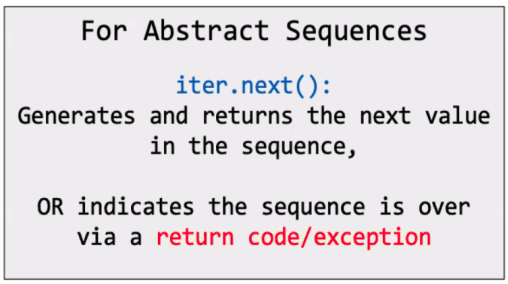
The iterator is separate from the iterable object that actually holds the sequences.

- **Iterable object**: a thing that has a bunch of values (a map, set, etc)
- **Iterator**: iterates over an iterable object

To create an iterator, you must ask the **iterable object** that **holds/defines** your sequence to give one to you.

While iterators differ by language, most tend to have an interface that look like one of these:





When you loop over the items in an iterable object, the language secretly uses an **iterator** to move through the items.

How Are Iterators Implemented?

1. Traditional Classes

An **iterator** is an object, implemented using regular OOP classes.

2. "True Iterators" aka **Generators**

   An iterator is implemented by using a special type of **function** called a **generator**.

Generators

They can be **paused** and **resumed**, with its state (variables, instruction pointer) saved between calls!

In python, you call the generator function with its parameters to create a generator object. Since a generator is an iterable object, and produces an iterator, you can use it like another iterable object in loops.

- Another name for a generator function is a **true iterator**.
- A generator function is really just an iterable object.

Iterator Objects and Generators

What can we do with an iterator object that we can NOT do with a generator, and vice-versa?

You can do everything with both! They are functionally equivalent in most instances.

Only Exception: Some iterators, like in C++, can move both forward and backward which can NOT be done with a generator.

Why would you prefer iterating using a generator vs iterating over a list of the same size?

A generator would save memory since a generator is just a closure with a couple local variables and an instruction pointer.

Iteration with First Class Functions

```
// Rust
(0..10).for_each(|elem| println!("{}", elem));
```

```
let items = vec!["fee","fi","fo","fum"];

items.iter().for_each(|elem| println!("{}", elem));
```

```
// Kotlin
(0..9).forEach({ elem -> println("$elem") })
```

```
var items = arrayOf("fee","fi","fo","fum")

items.forEach({ elem -> println("$elem") })
```

Here, we are passing a function as an argument to a forEach0/each() method that loops over the iterable's items.

Lecture 18: Control Palooza, Part 2: Concurrency and Multithreading

Concurrency

Many modern programming languages have explicit support for concurrent execution (running multiple things at once).

Concurrency is a paradigm in which a program is decomposed into simultaneously executing tasks.

Those tasks might…

1. Run in parallel on a single core
   OR
   Run **multiplexed** on a single core

   Definition: *Multiplexed* means combining and sending multiple data streams over a single medium/stream.

2. Operate on totally independent data (e.g., each sorting a different array)
   OR
   Operate on shared *mutable* data or systems (e.g., two tasks modifying a shared queue)

3. Be launched deterministically as part of the regular flow of a program
   OR
   Be launched due to an external event occurring, like a click of a button in a UI

Concurrency can make our programs more efficient!

The old way: **Serial Execution**

The new way: **Concurrent Execution**

- It could run in parallel like this or in a single thread.

If a concurrent program only runs on a single core, it could be faster than a program that runs serially because some tasks involve lots of waiting, but make little use of the CPU.

Concurrency is when:

- The execution of multiple tasks is interleaved, instead of each task being executed sequentially one after another.

It is:

- The approach of executing different parts of a program out-of-order (possibly in parallel), to yield the same result more efficiently than serial execution.

- The execution of the multiple instruction sequences at the same time.
- The execution of multiple computations simultaneously
- A software structuring technique that allows us to model computations as hypothetical independent activities (e.g. with their own program counters) that can communicate and synchronize.

A concurrent program has multiple threads or tasks of control allowing it to perform multiple computations in parallel and to control multiple external activities that occur at the same time.

Getting concurrency right is challenging because each thread of execution can be interrupted at any time to run the other.

- Any time two threads of execution use a "shared mutable state", this is called a **race condition**:
- A **race condition** or race hazard is the condition of an electronics, software, or other system where the system's substantive behavior is dependent on the sequence or timing of other uncontrollable events. It becomes a bug when one or more of the possible behaviors is undesirable.

    A key goal of concurrency is *determinism*: ensuring that code produces the same result every time it runs (regardless of how its tasks are scheduled)!

<u>Concurrency With Shared Mutable State</u>

Most modern languages now have built-in language features to make it safer to use shared mutable state.

Java has a **Synchronized** keyword that can be used to limit access to a **mutable variable** to a single thread at a time.

<u>Concurrency Approaches</u>

High-level approaches to concurrency in modern languages.

Full Multithreading

A program may launch any number of threads, and their execution is interleaved across available cores (C++, Java, C#, Kotlin, Rust, etc).

Limited Multithreading

The language supports concurrent execution, but there are constraints preventing some kinds of parallelism (Python).

Simulated Multithreading

The language supports concurrent execution, but all execution is actually limited to a single thread (JavaScript, TypeScript, etc).

Models for Concurrency

Multi-Threading Model

- A program creates multiple "**threads**" of execution that run concurrently (potentially in parallel).
- The programmer explicitly "**launches**" one or more functions in their own threads, and the OS schedules their execution across available CPU cores.

Event Driven Model

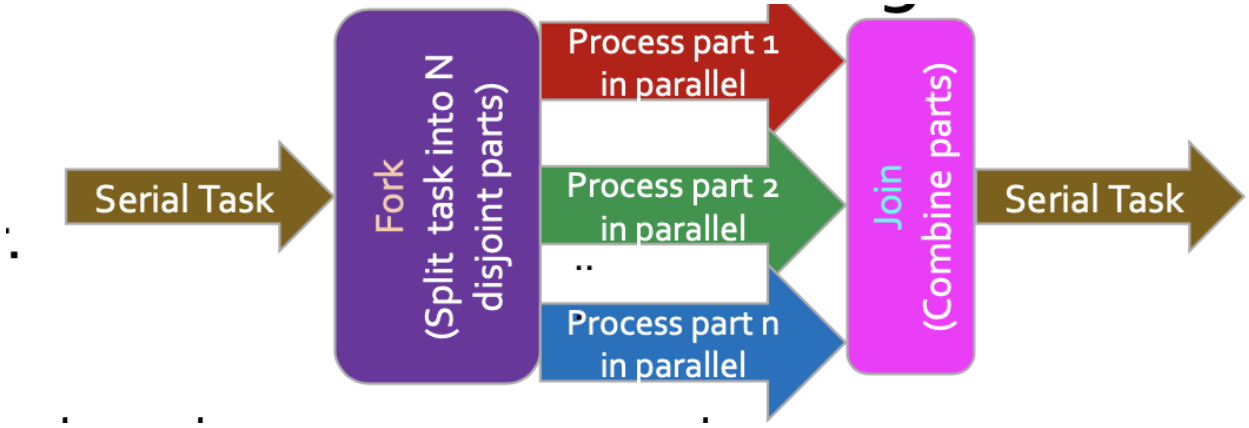
- A program consists of a queue of functions to run, and an infinite loop that dequeues and runs each function from the queue, one after the other.
- When an **event** occurs (e.g., user clicks a button) the event results in a new function f() being added to the queue which eventually runs and **handles the event**.

Hybrids: You can launch background threads in event-driven systems, for example.

Multi-Threading

Fork-Join: A Common Pattern for Multi-Threading

The "fork-join" pattern is basically a concurrent version of divide and conquer.



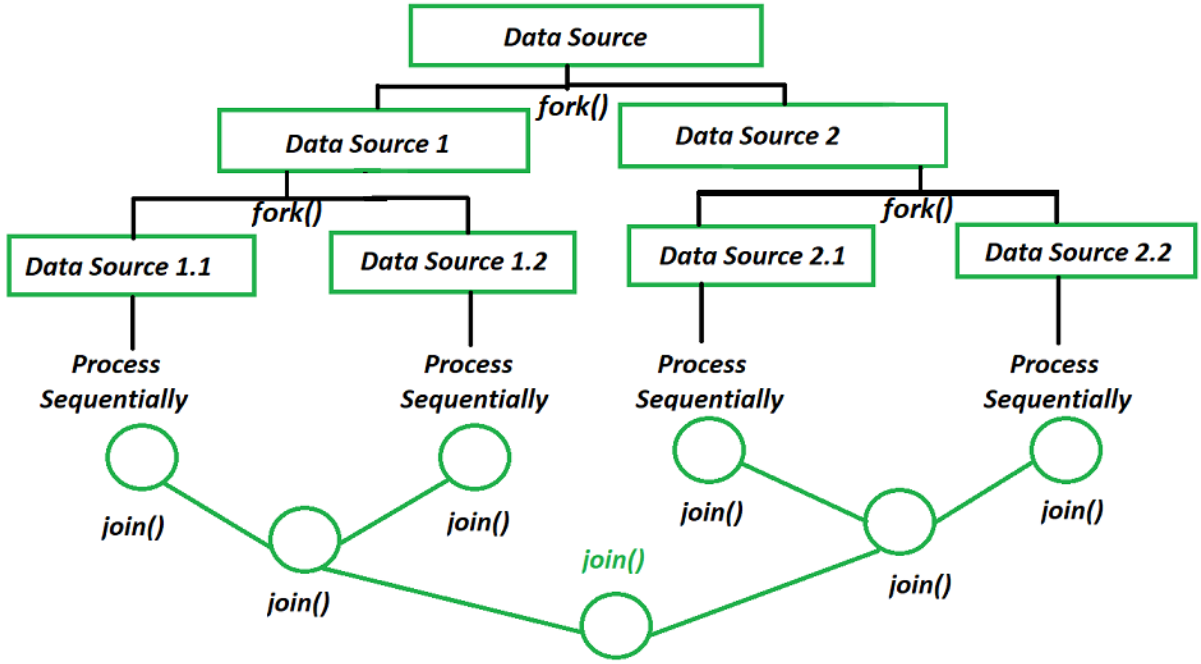
First, we "**fork**" one or more tasks so they execute concurrently...

Second, we wait for all those tasks to complete (aka "**join**") and then proceed.

So other tasks can run in this computational void.

Generally, waiting doesn't actually take any CPU power…

Fork-Join is Often Used Recursively



Fork-Join in Different Languages: C++

```
// C++
#include <thread>

void task1(int n)  { ... }
void task2(int n)  { ... }
void task3(int n)  { ... }

int main() {
  std::cout << "Forking threads!\n";
  std::thread t1(task1, 10);
  std::thread t2(task2, 20);
  std::thread t3(task3, 30);

  // do other processing here...

  t1.join();
  t2.join();
  t3.join();
  std::cout << "All threads joined!\n";
}
```

Fork-Join in Different Languages: Python

```
# Python
import threading

def task(n):
  while n > 0:   # do some computation
    n = n - 1

print("Forking threads!")
t1 = threading.Thread(target=task, args=(100000000,))
t2 = threading.Thread(target=task, args=(100000000,))
t3 = threading.Thread(target=task, args=(100000000,))
t1.start()
t2.start()
t3.start()

# do other processing here...
t1.join()
t2.join()
t3.join()
print("All threads joined!")
```

Assuming looping 100 million times takes 5s, it will take this program 15s to run on a multicore PC.

Because when each Python thread runs, it claims exclusive access to Python's memory/objects.

- So only one thread generally does computation at a time

Python's garbage collection system was never designed to be thread-safe!

Python has something called a **Global Interpreter Lock** or **GIL**.

The GIL is like a hot potato – it can only have one owner at a time.

Once a thread takes ownership of the GIL it can read/write to Python objects.

- After a thread runs for a while, it releases the GIL (tosses the potato) to another thread and gives it ownership.
- If a thread is waiting for the GIL, it simply falls asleep until it gets its turn.

Why even support multiple threads? Does it ever speed things up?

I/O operations like downloading data from the web or saving a file to disk DON'T need the GIL to run!

- So these kinds of operations can still make progress if launched in the

background!

- In addition to I/O ops, many external C++ libraries (e.g., pytorch) have operations that can also run in parallel!

*Overall, **Javascript** only has 1 thread and **Python** can have multiple threads and can support multithreading, but it can NOT have parallelism that supports compute-bound code due to the Global Interpreter Lock (GIL)*

Definition: *CPU (Compute) Bound* describes a program that would go faster if the CPU were faster (i.e. it spends the majority of its time using the CPU – doing calculations. This could be crunching numbers by computing new digits of PI).

Definition: *I/O Bound* describes a program that would be faster if the I/O subsystem was faster. This is usually associated with a task that processes data from the disk (for example: counting the number of lines in a file).

Coroutines

A coroutine is one of several procedures that take turns doing their job and then pause to give control to the other coroutines in the group. (C#, Python, Ruby, PHP, JavaScript, Rust, ...)

Coroutines are computer program components that generalize subroutines for non-preemptive multitasking, by allowing execution to be suspended and resumed.

- In the case of threads, it's an operating system (or run time environment) that switches between threads according to the scheduler. While in the case of a coroutine, it's the programmer and programming language which decides when to switch coroutines. Coroutines work cooperatively multitask by suspending and resuming at set points by the programmer.

Multiprocessing

Definition: Usually refers to many processes executed in parallel. It is more reliable than multithreading since if one process crashes, it won't affect other processes. In contrast, if one thread crashes, the process it belongs to will also crash as well as other threads in the same process.

It is good for **isolated** tasks including computing a page-rank from 3 different sources in parallel. Other examples include: Spark, Hadoop, and Distributed Computing.

Cooperative Multitasking, Coroutines

- A coroutine is a subprogram that has multiple entries and controls them itself.
- Coroutines provide quasi-concurrent execution of program units (the coroutines); their

execution is interleaved, but not overlapped.

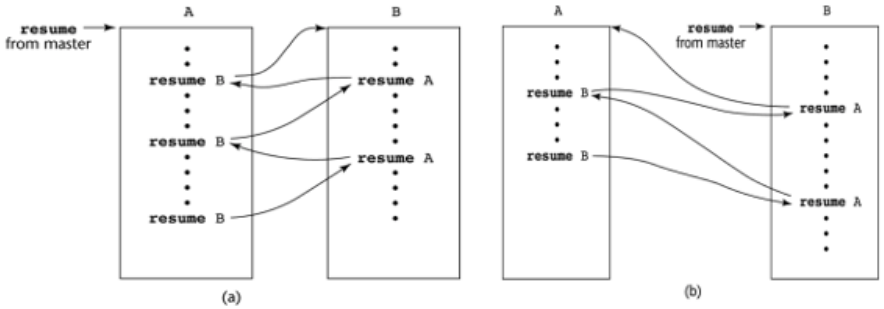


Figure 9.3 Two possible execution control sequences for two coroutines without loops

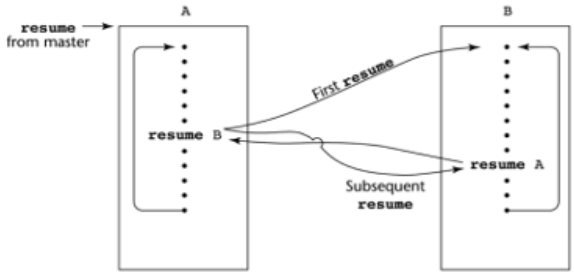


Figure 9.4 Coroutine execution sequence with loops

UI Programming and Concurrency

When we're building user interfaces, concurrency works a bit differently

It might be surprising, but most UIs (Windows, MAC, iPhone, Android, browsers) are single-threaded and "**event-based**."

Callback Functions

A callback function is a function that is called when an "**event**" occurs.

The event could be a click of a UI button, a timer expiring, or a completed payment Transaction.

- The callback performs a follow-up action that handles the just-completed event.

For example, a callback handling a button click could initiate a credit-card purchase… and the callback that's triggered when the purchase completes would update the webpage with the payment result.

Web Browser in a Slide

While traditional programs run from top-to-bottom, programs with UIs use a different model.

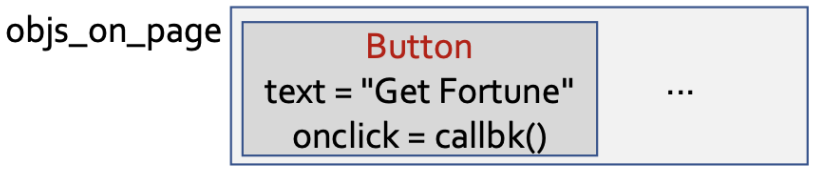
This is because UIs contain elements (e.g., buttons, sliders) that the user can interact with in any order!

So UIs use something called an **Event Loop** to control their execution.

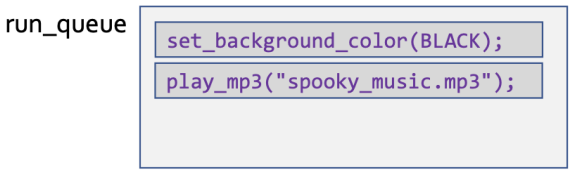
Let's look at a high-level model for a web browser to see how the Event Loop works:

The browser manages two sets of items:

- It manages all the objects on a page (e.g., buttons, sliders, etc.).



- It maintains a queue of statements to execute from the page.



Now let's see our event loop – it runs an infinite loop that does two things.

1. It checks to see if the user interacted with the page's elements, and if so, deals with this.

2. It gets the next statement from the queue (if there are any) and runs it.
   The loop runs over and over, handling events, and running the next queued statement…

How do we handle user-interaction events?

We cycle through each of the UI elements on the screen to see if any have been interacted with…

If an element has experienced an event (e.g., a **click**)... we ask it for its handler (aka **callback**) function… But we DON'T call the handler immediately!!!

- Instead, we queue the function up for later execution!

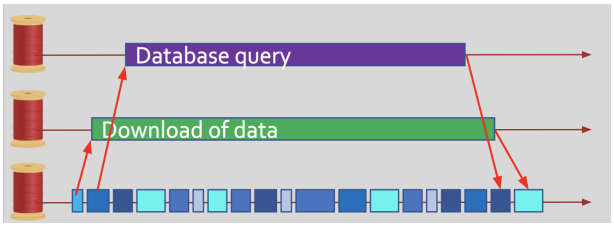
Finally, let's see how we run statements.

We just check to see if the queue has pending statements to run… and if so, we dequeue the next statement and run it to completion!

- The **Event Loop** runs over and over…
  1) Checking for new events…
  2) Potentially adding items to the run queue…
  3) Then running the next item in the run queue until completion…

But wait, does ALL browser activity run in the event loop?

- By default, all user-supplied UI code (e.g., **Javascript**) DOES run on the event loop… (so all on a *single thread*)
  But things like I/O (e.g., downloading data, querying a **Database**) DO run on *background (multiple) threads*!
  - I/O is also tracked by the event handler just like UI events!
  - When they finish, the event handler queues up a callback to run on the main **thread**.
  - In addition, you can also launch your own background threads, but there are caveats.



Event Loop Challenge

What happens if one of the statements in the run queue is "CPU bound" and takes a **long time** to run?

The event-loop can only do one thing at a time, so if a function (or any of the functions it calls) takes a long time to run…

- The event handler can't run and process UI events like clicks, etc.
- This causes the UI to hang and become unresponsive to clicks, etc.

For browsers, tabs generally get their own **thread**, so that one tab will not block the others.

Every tab is generally treated as a separate **process**, so they are isolated from each other and they each have their own event loop.

Event Loops Are Everywhere

Mobile App Frameworks
Backend Frameworks (Node.js)
Industrial Robotics
Mars Rover

Event Loops and Sequences of Callbacks

While we saw how to run a long-running op in the background and use a **callback** upon completion to perform the next processing step, sometimes we'll just need to run a single background op…

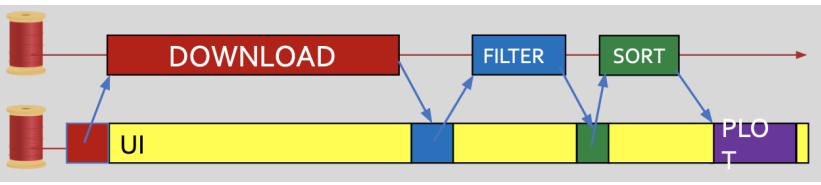
In many cases, we want to chain multiple long-running background ops together into a **pipeline**.

- This lets us accomplish more complex tasks in the background.

Chaining Background Ops Together

Let's say we want to download some data, then **filter** the data, then **sort** the data, then **update** the UI.

We'd like to perform these tasks in the background, while the event loop lets the user interact with the UI…



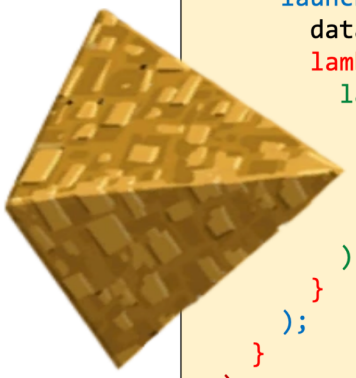
To make this happen, we provide each phase of the pipeline with a callback function that indicates the next function to run.

Chaining (With Callbacks) Can Get Ugly

Without going into detail, the syntax to chain multiple background functions together with callbacks is pretty ugly and requires the use of lambda functions…

This nesting syntax gets so ugly that it's been given a few names including…

- **Callback Hell** and **The Pyramid of DOOM**!



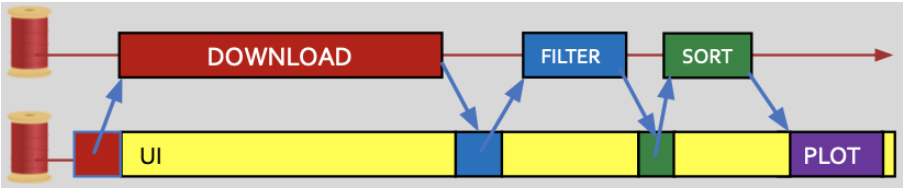
```
function display_sorted_filtered_data(url) {
  launch_download(
    url,
    lambda (data) {
      launch_filter(
        data,
        lambda (fdata) {
          launch_sort(
            fdata,
            lambda (sdata) {
              display_in_ui(sdata);
            }
          );
        }
      );
    }
  );
}
```

Promises: A Cleaner Syntax for Chaining Operations

In our previous example, we first download a file, and then filter the data, and then sort the remaining data, and then plot the data in the UI.

Wouldn't it be nice if we could use a simpler syntax than callbacks? How about this?

Each phase of the "pipeline" does its work, produces a result, and passes that result to the next lambda in the pipeline.



This syntax hides what's really going on – it's really just using callbacks like before!
When you create a pipeline like this, it doesn't run synchronously... it runs concurrently!

Promises and Errors

To deal with errors that occur in the pipeline, we use an "exception-like" syntax – but it's not an exception!

```
function display_sorted_filtered_data(url) {
  download_data(url)
    .catch(lambda (error)           { deal_with_dl_error(error);      } )
    .then(lambda (downloaded_data) { filter_data(downloaded_data);  } )
    .catch(lambda (error)           { deal_with_filter_error(error); } )
    .then(lambda (filtered_data)    { sort_data(filtered_data);       } )
    .catch(lambda (error)           { deal_with_sorting_error(error);} )
    .then(lambda (sorted_data)      { update_ui(sorted_data);         } )
}
```

If you don't catch something in-line, you can also catch errors from any phase of the pipeline at the end.

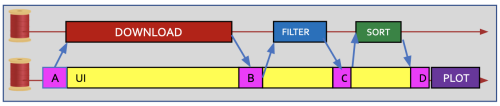
- Although this provides less error-handling granularity, so be careful.

Async/Await Syntax for Chaining Operations

The async/await model is yet another approach with even simpler syntax.
Here's the async/await version of our earlier pipeline in JavaScript:

```
async function download_data(url) { ... }
async function filter_data(data) { ... }
async function sort_data(data) { ... }
function update_ui(data) { ... }

async function display_sorted_filtered_data(url) {
  data = await download_data(url);  // A
  fdata = await filter_data(data);  // B
  sdata = await sort_data(fdata);   // C
  update_ui(sdata);                 // D
}
```



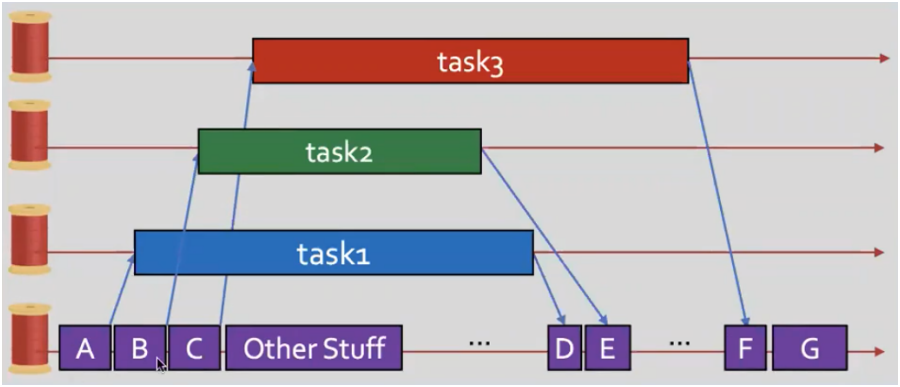
async means that the function will run asynchronously and deliver its result at some unknown point in the future.

When you use this paradigm, there are two phases for executing tasks:

1) **Launch the Async Function**: You request execution of a function in the background and obtain a handle to the pending task.
    - The async function immediately returns a "handle" that can be used to track its progress.
    - The handle is NOT the function's return value, but an object we can use to get the return value when the function eventually finishes running.

2) **Wait for Completion**: You may optionally wait for your launched task to complete its execution, and then obtain its result
    - Await suspends execution of our current function until the **async task** finishes (and returns a result).

Now, there's no reason you must immediately await tasks like in our previous example:



Lecture 19: Logic Palooza: Logic Programming

Logic Programming

A paradigm where we express programs as a set of faces (relationships which are held to be true), and a set of rules (if A is true, then B is true)

Given **Facts** and **Rules**:

- A program/user then issues queries against these facts and axioms?

Logic programming is declarative – programs specify "**what**" they want to compute, not "**how**" to do so.

Prolog: A Logic Programming Language

It is used for theorem proving (type checking is one such use case) and has also been used to implement Natural Language Processing (NLP) and expert systems.

It is primarily declarative!

When given a query, Prolog tries to find a chain of connections between the **query** and the specified **facts** and **rules** that lead to an answer.

Prolog Implementations

There are two major implementations of Prolog: GNU and SWI

Prolog Programs

Every Prolog program consists of **facts** about the world, and a bunch of **rules** that can be used to discover new facts.

A fact is a predicate expression that declares either…

- An attribute about something (aka **atom**):
- A relationship between two or more **atoms** or **number**

Facts look like function calls, but they are really just static assertions of different relationships.

In Prolog, the **functor** and all **atoms** must be lower case.

Facts can also be nested arbitrarily deep!

- If a fact is not explicitly stated, then it is assumed to be false.

A **rule** lets us define a new fact in terms of one or more existing **facts** or **rules**.

Each rule has two parts:

- A "**head**" which defines what the rule is trying to determine

- A "**body**" which specifies the conditions (aka subgoals) that allows us to conclude the head is true.

Syntax:
- ":-" = if
- A comma (,) separating the parts of a rule means "AND" in Prolog

Rules are defined in terms of **Variables** as well as **atoms** and **numbers**.
- Variables like X and Y are placeholders, which Prolog will try to fill in as it tries to answer user queries.
- Variables must always be **Capitalized**.

Rules can also be recursive and can have multiple parts:

```
% Recursive rules:
ancestor(X, Z) :- parent(X, Z) % base case
ancestor(X, Z) :- parent(X, Y), ancestor(Y, Z) % inductive case
```

Just as with recursion, always put the base-case first, since when answering queries, Prolog processes rules from top to bottom

Rules can also use **negation** – but be careful!

```
% Rules with negation:
serious(X) :- not(silly(X)) % base case
```

In Prolog, not(<something>) works as follows:
1. Prolog tries to prove <something> is true using all of the program's facts and rules.
2. If <something> can NOT be proven is true, then not <something> is found to be true.

Prolog operates according to the **Closed World Assumption** (CWA).

The CWA states that only those things can be proven to be true by the program facts/rules are true.

<u>Prolog Queries</u>

Once we define our facts and rules, we can compile our program and **query** them.

There are two types of queries
1. We can create queries to answer **true/false questions**
2. We can aso create queries to **fill-in-the-blanks**.

In Prolog, the process of coming up with answer(s) to a query is called **Resolution**.

Let's See How Prolog Answers Queries: Resolution

Prolog's Resolution Algorithm

We'll use a simpler database of facts and rules for this:

```
parent (nel, ted).
parent (ann, bob).
parent (ann, bri).
parent (bri, cas).
gparent( X ,  Y )  :-
        parent(X,Q), parent(Q,Y).
```

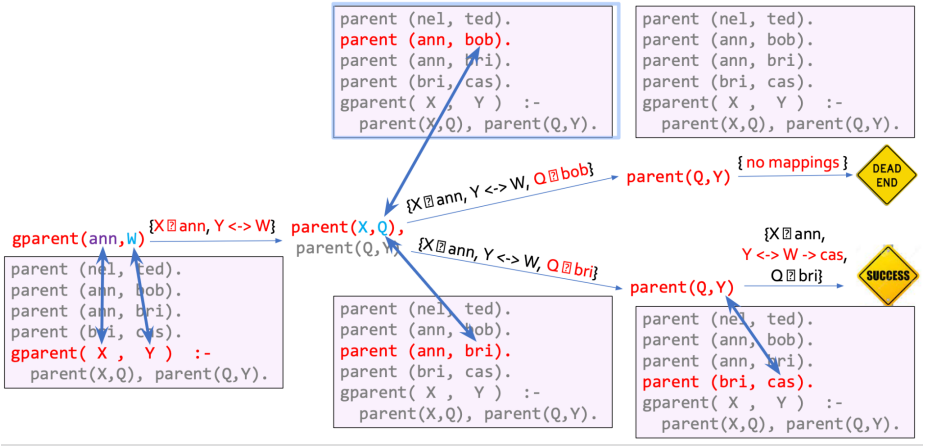
Let's answer: Who is ann a grandparent of?

```
gparent(ann, W)
```

1. Add our query to a goal stack
2. Pop the top goal in our goal stack and match it with each item in our database, from top to bottom
3. If we find a match, extract the variable mappings, and create a new map with the old+new mappings
4. Create a new goal stack, copying over existing, unprocessed goals and adding new subgoals
5. Recursively repeat the process on the new goal stack and the new mappings

Prolog goes all the way down until there is nothing left!

Visualizing Prolog Resolution as a Tree



The Resolution Algorithm: Unification

As we saw, during the Resolution process, Prolog repeatedly compares the current **goal** with each **fact/rule** to see if they are a match.

If a **goal** and a **fact/rule** match, Prolog extracts mappings between variables and atoms (e.g., X → bob).

- Together, these two steps (comparison + mapping) between a single goal and a single fact/rule are called **unification**.
- The unification algorithm runs on one goal at a time.
- Unification runs on a single fact or rule at a time.

Here's the unification pseudocode:

```
def unify(goal, fact_or_rule, existing_mappings):
  if the goal with the existing mappings matches the current fact/rule:
    mappings = extract variable mappings between goal and fact/rule
    return (True, mappings)     # We did unify! Return discovered mappings
  otherwise:
    return (False, {})          # We couldn't unify! So no mappings found
```

Unification: Matching a Goal and a Fact/Rule

A goal with the current mappings matches a specific fact/rule based on these steps:

Step 1: Apply all current mappings to the goal.

Step 2: Treat both the mapped goal and head of the fact/rule as trees.

Step 3: Compare each node of the goal tree w/the corresponding node in the fact/rule tree.

- If both nodes (Goal with mappings applied and the Fact/Rule) are functors, then make sure the functors are the same and have the same number of children (aka **arity**).
- If both nodes hold atoms, make sure the atoms are the same.
- If a goal node holds an unmapped variable, it will match ANY item in the corresponding node of the fact/rule.
- If a fact/rule node holds an unmapped variable, it will match ANY item in the corresponding node of the goal.

Unification: Extracting Variable Mappings

Once we know that a goal matches a fact/rule, we need to extract the variable mappings.

Here's the approach:

- Iterate through each corresponding pair of nodes in both trees.
- Any time you find an unmapped Variable in either tree that maps to an atom/number in the other tree, create a mapping between the variable and the atom.
- Any time you find an unmapped Variable in either tree that maps to an unmapped Variable in the other tree, create a bidirectional mapping between the variables.

Unification: Summary

Unification is the process of comparing a single goal with a single fact/rule, given the current set of mappings and determining if the goal and the fact/rule match, and … if so, extracting all new mappings between variables and atoms on either side.

Unification is used within the broader Resolution algorithm that we traced through.

The Full Resolution Algorithm: Pseudocode

```
def resolution(database, goals, cur_mappings):
  if there are no goals left:
    tell the user we found a solution and output our discovered mappings!
    return
  for each fact/rule z in the database:
    success, new_mappings = unify(goals[0], fact_or_rule, cur_mappings)
    if success:
      tmp_mappings = cur_mappings + new_mappings
      tmp_goals = sub_goals(z) + goals[1:]
      resolution(tmp_goals, tmp_mappings)    # recursion
  # if we get here, we didn't find a match... BACKTRACK and keep trying!
```

The Resolution algorithm is initially called with the user's query as its only goal, and with no initial mappings:

```
# Determine if ann is the grandparent of cas?
resolution(database, "gparent(ann, cas)", { })
```

Prolog Lists

Lists in Prolog are just like lists in Haskell or Python.

Lists can contain numbers, atoms, or other lists.

```
[ ]
[silly, goofy, gleeful]
```

[1, 2, [dog, cat], 3.5]

A statement is a unit of program execution that expresses an action to be carried out.
Statements generally don't result in a value – they simply carry out an action.

Prolog uses a combination of **pattern matching** and **unification** to process lists.
List processing is also done with facts and rules, just like either inference tasks.

Prolog has **Variable(s)** instead of **atoms**.

Prolog List Processing

is_head_item(X,[X | XS]).

This has a similar syntax to Haskell's pattern matching with (x:xs). X matches the first item of the list and XS matches the rest of the list.

- Overall, Prolog is unifying from left-to-right and mapping each variable.
- Once it extracts a mapping, it only "unifies" the query if later uses of the mapping are consistent with the first.

is_second_item(Y,[X, Y | XS]).

This has a similar syntax to Haskell's pattern matching with (x:y:xs). X matches the first item of the list, Y matches the second item of the list, and XS matches the rest of the list.

is_member(X,[X | Tail]).
is_member(Y,[Head | Tail]) :- is_member(Y, Tail).

Now, we are checking if a list contains a value. This one consists of a fact and a recursive rule.

- The first fact checks if a value X is the first item in the list.
- The second part (rule) uses pattern matching to break up the list into the **Head** item and a list of all the **Tail** items.

  And then it attempts to prove that value Y was found somewhere in the Tail list.

delete(ItemToDelete, ListToDeleteFrom, ResultingList).
delete(carey, [paul, carey, david], X) → X = [paul, david]

Now: Deleting a single item from a list

```
delete(Item, [Item | Tail], Tail).
delete(Item_, [Head_ | Tail_], [Head_ | FinalTail]) :- delete(Item_, Tail_, FinalTail)
```

The first fact is like we saw earlier – it handles the base case. The base case handles the situation where the first item in the list is the one we want to delete, e.g.: delete(carey, [carey, david, paul], X).

- We're trying to delete an item with this value.
- So we break up the list into the head Item and all Tail items using pattern matching.
- And verify that the head item in the list is the same as the item we want to delete.
- If so, our final list will just be the Tail of the list – it contains all items except the deleted one!

The second line handles the case where the item we want to delete is NOT the first item.

- Our rule uses pattern matching to break up the input list into the Head Item/all Tail items.
- The subgoal (after the "**:-**") says this: "Use delete to remove the Item from amongst the Tail items; FinalTail refers to the resulting tail."
- The third parameter which is "**[Head_| FinalTail])**" represents our final list with the item deleted.

  In this case, we construct our output list by concatenating the Head item from our original ist with the tail of the list with the Item removed from it.

<u>Proof List Processing: Built-In Facts and Rules</u>