C++, Ruby, CoffeeScript: a visual comparison of language complexity

Jun 7, 2012 •

Most people will agree that C++ is a fairly complex language. But just how complex is it? I got curious about quantifying that by comparing the number of concepts a programmer has to understand to learn a programming language in its entirety.

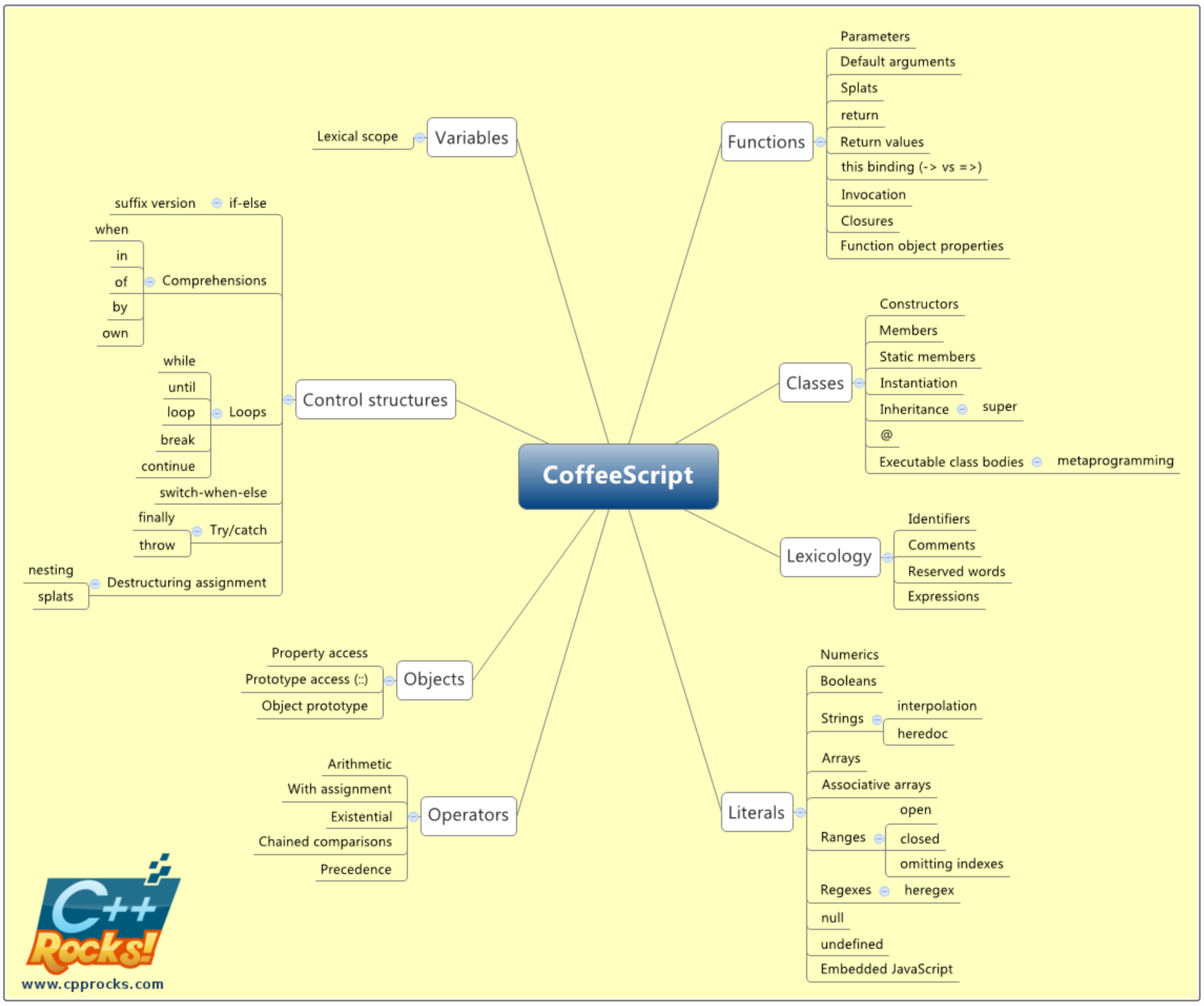
Concepts in my definition include large categories like classes and operators, as well as more fine grained things like *if-else* statements and *while* loops. There is a lot of variation in the complexity of different concepts, so their number can only serve as a rough measure of language complexity. Nevertheless, I think it's interesting.

I chose C++, CoffeeScript and Ruby for my comparison. CoffeeScript and Ruby are dynamically typed so they are significantly different from C++. However, all three are multi-paradigm general purpose languages, supporting (to a reasonable degree at least) object oriented, functional, procedural and generic programming. So this post is about C++ vs. dynamically typed languages.

I think it would also be interesting to make a comparison with other statically typed languages such as F# or Scala but I'm not really familiar with them, so it would be hard for me to do.

CoffeeScript

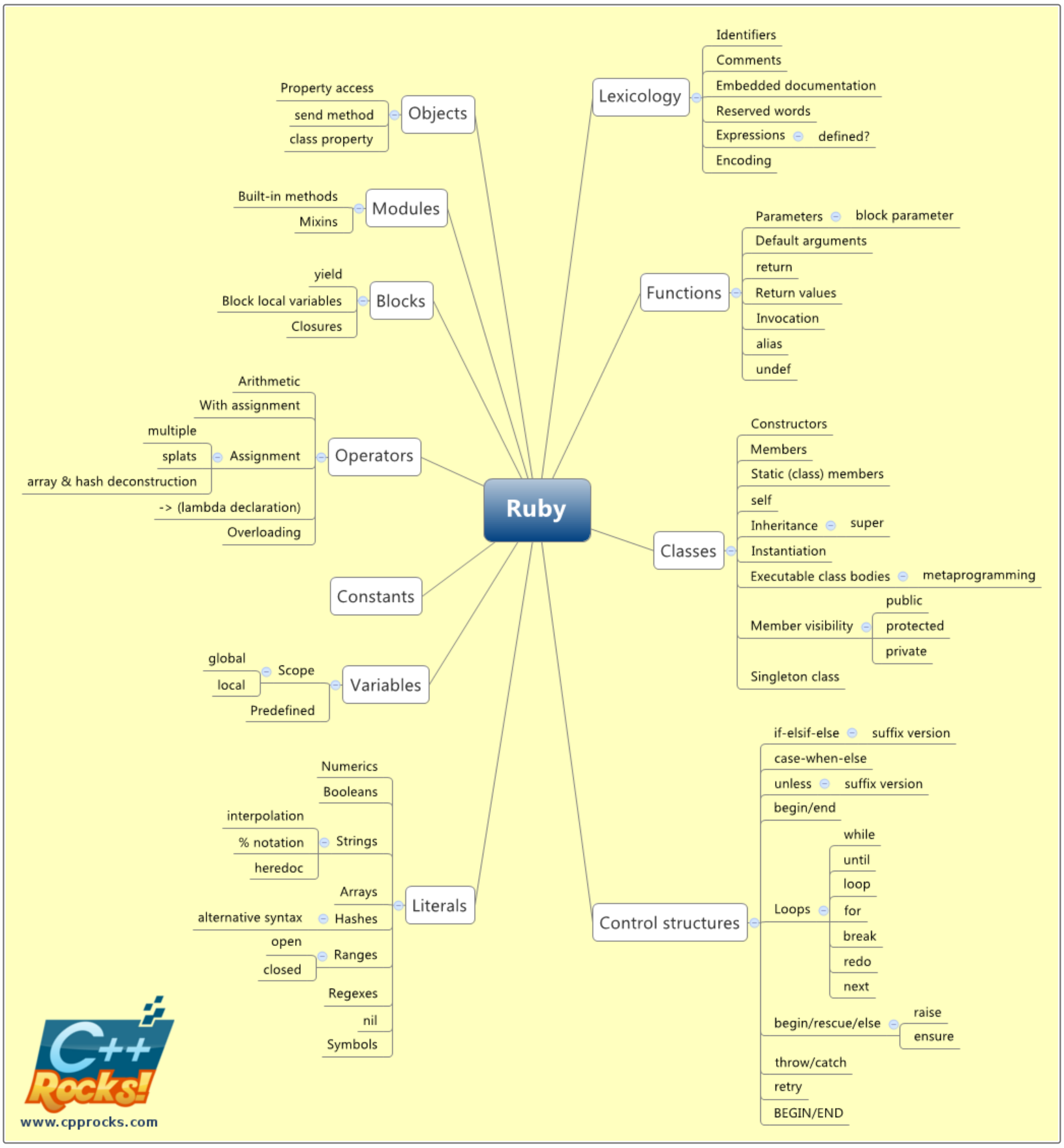
First up, let's look at CoffeeScript:



It has a total of **68** concepts divided into **8** major groups.

Note that I listed metaprogramming as one concept because it's largely based on the concept of executable class bodies.

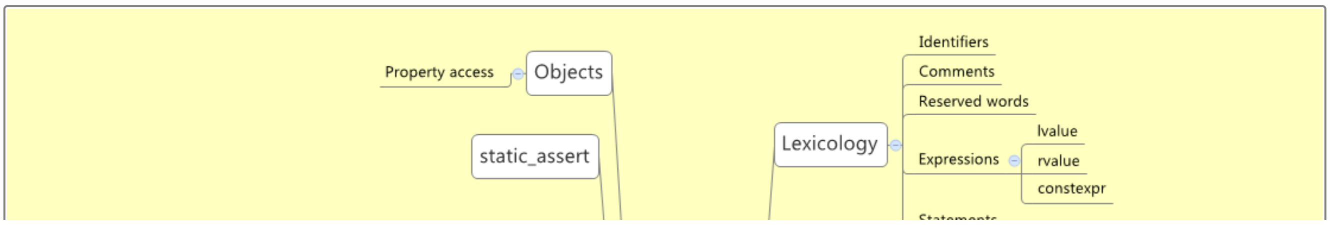
Ruby

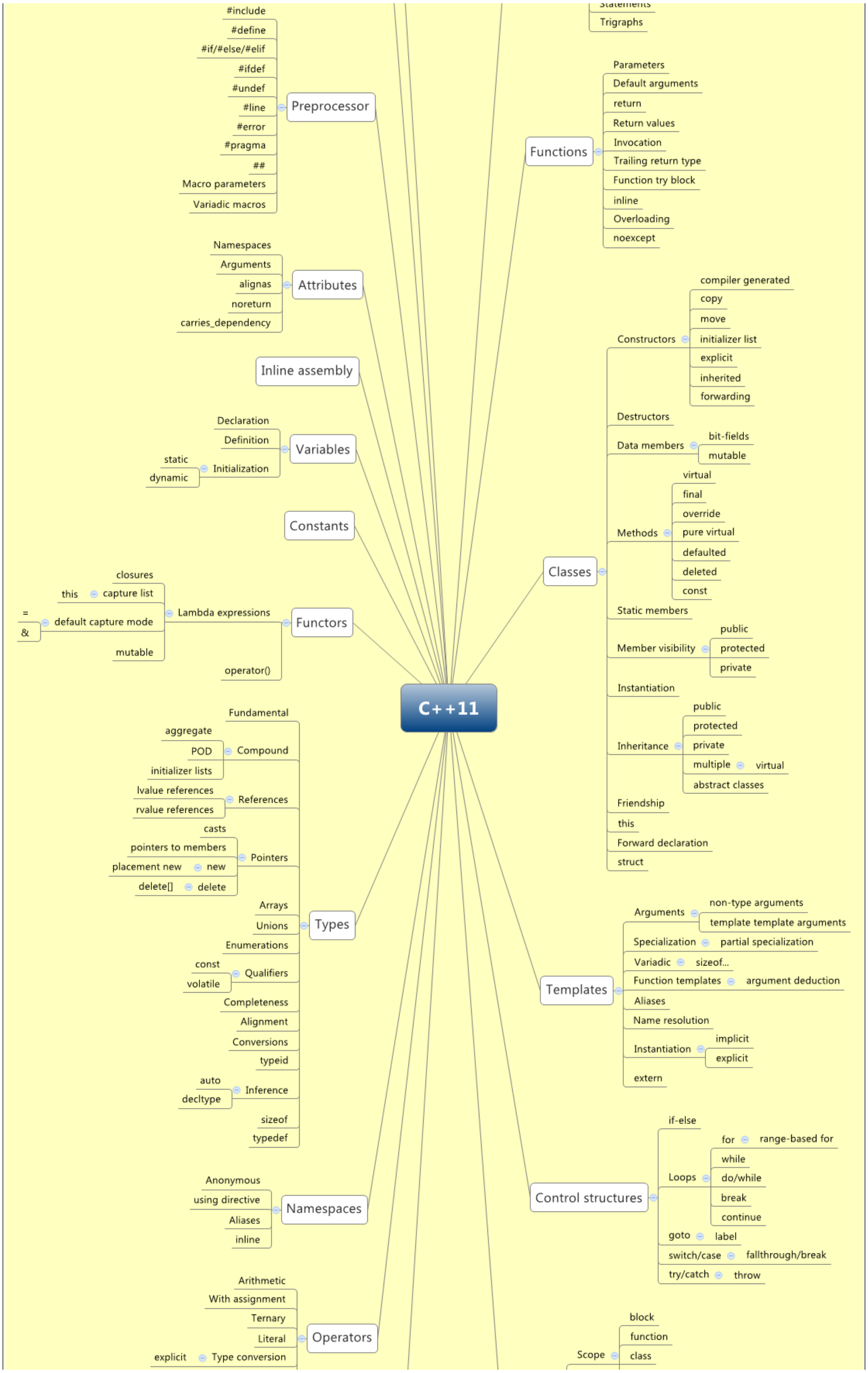


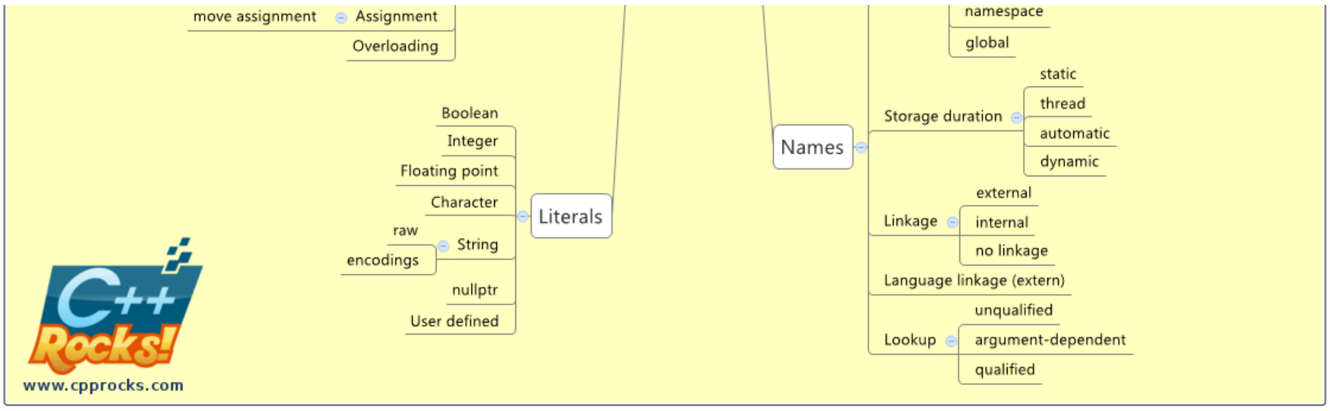
Ruby is clearly more complicated, with **96** concepts in **11** major groups. It’s got a more sophisticated class model than CoffeeScript, as well as things like constants, blocks and operator overloading.

C++

Finally, here is C++:




C++11
Preprocessor
#include
#define
#if/#else/#elif
#ifdef
#undef
#line
#error
#pragma
##
Macro parameters
Variadic macros
Attributes
Namespaces
Arguments
alignas
noreturn
carries_dependency
Inline assembly
Variables
Declaration
Definition
Initialization
static
dynamic
Constants
Functors
Lambda expressions
closures
capture list
this
default capture mode
=
&
mutable
operator()
Types
Fundamental
Compound
aggregate
POD
initializer lists
References
lvalue references
rvalue references
Pointers
casts
pointers to members
new
placement new
delete
delete[]
Arrays
Unions
Enumerations
Qualifiers
const
volatile
Completeness
Alignment
Conversions
typeid
Inference
auto
decltype
sizeof
typedef
Namespaces
Anonymous
using directive
Aliases
inline
Operators
Arithmetic
With assignment
Ternary
Literal
Type conversion
explicit
Statements
Trigraphs
Functions
Parameters
Default arguments
return
Return values
Invocation
Trailing return type
Function try block
inline
Overloading
noexcept
Classes
Constructors
compiler generated
copy
move
initializer list
explicit
inherited
forwarding
Destructors
Data members
bit-fields
mutable
Methods
virtual
final
override
pure virtual
defaulted
deleted
const
Static members
Member visibility
public
protected
private
Instantiation
Inheritance
public
protected
private
multiple
virtual
abstract classes
Friendship
this
Forward declaration
struct
Templates
Arguments
non-type arguments
template template arguments
Specialization
partial specialization
Variadic
sizeof...
Function templates
argument deduction
Aliases
Name resolution
Instantiation
implicit
explicit
extern
Control structures
if-else
Loops
for
range-based for
while
do/while
break
continue
goto
label
switch/case
fallthrough/break
try/catch
throw
Scope
block
function
class



There are **189** concepts in **18** groups on this diagram, double the number of Ruby concepts and almost 3 times more than in CoffeeScript!

I believe that some of the concepts in C++ (such as name resolution) are also comparatively more complex, with a lot of nuanced rules. The concept of names in C++ is more involved than in the other languages because there are many different categories. In dynamic languages, more or less everything is either a constant or a variable (including class names and function names).

I omitted the concepts of compilation and linking because the other two languages don't have them.

From the diagram we can see that there are many reasons for the complexity. Part of it is the static type system, another part is templates, yet another part is the name system, as well as the complex class model. The underlying drivers are of course performance and type safety.

What's not in the diagrams?

I didn't include a lot of other things that a programmer is required to know in order to use a language in practice:

- standard libraries
- language idioms and best practices
- concurrency concepts
- design patterns

These things become more prominent when a programmer progresses from learning to mastering a language.

I think that C++ would win the prize for complexity in such a comparison too. For example, there are books like *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices* (there are in fact 101 rules in there), and *Effective C++* containing 55 pieces of advice. These books don't deal with esoteric situations, they are aimed at normal everyday code. That's a lot of extra knowledge!

While other languages have their best practices and idioms too, I think their number is quite a lot smaller.

So what's the point?

My goal isn't to complain about how huge and incomprehensible C++ is. I believe there is no alternative to C++ for large projects with strict hardware constraints or performance requirements when you take the availability of modern tools, libraries and developers into account.

However, I do think that it's important to highlight what programming in C++ entails because it has implications for how it's taught and when it should be chosen for a project.

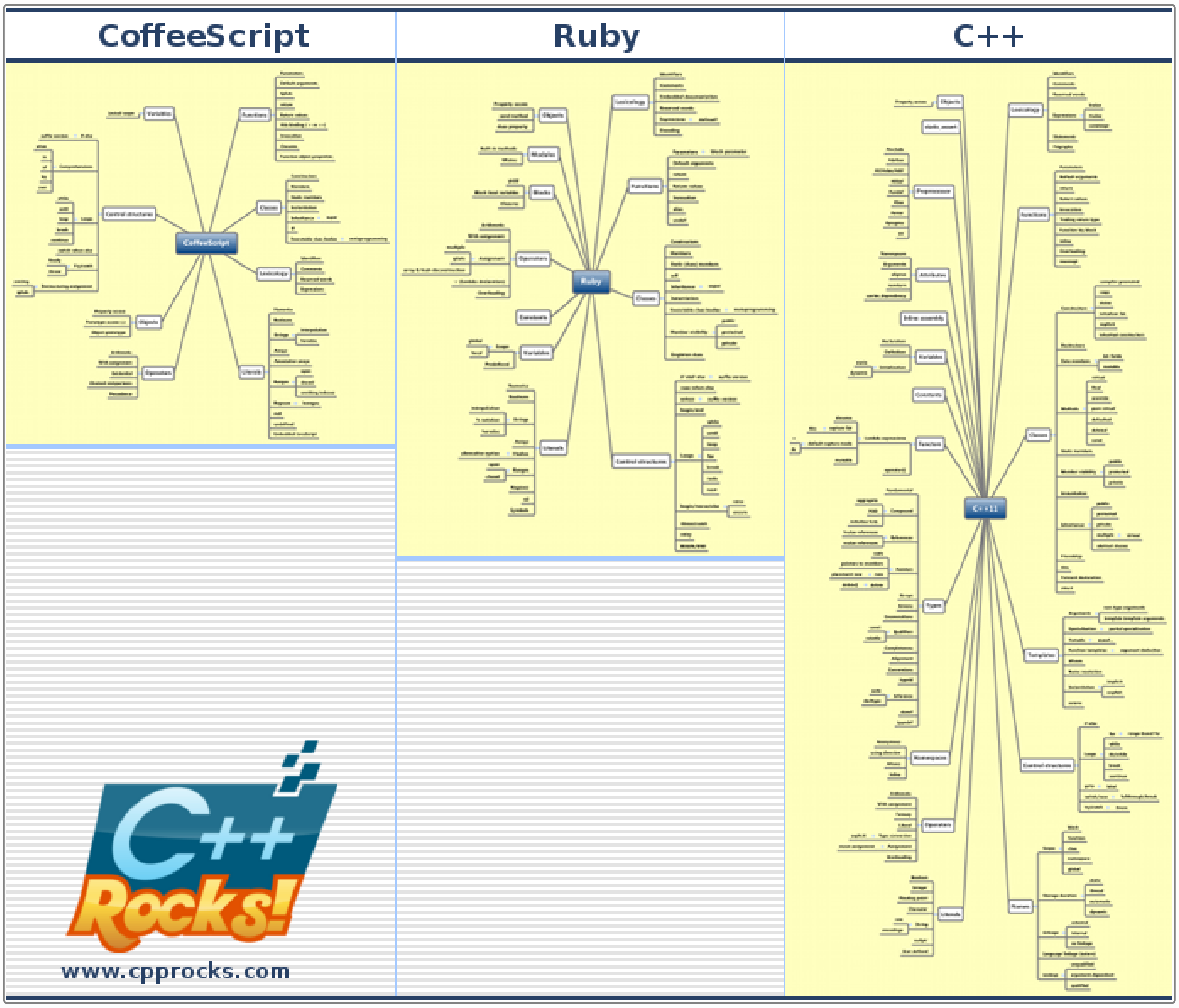
It isn't a language that can be picked up quickly. It takes a long time to learn and even longer to master. Because of this, I think there is a lot of value in learning C++ "from the top down".

With C++11, it has become easier to start with the modern high level subset of C++ (and the standard libraries), and then gradually add on the intricacies and the low level features. In other words, *std::shared_ptr* comes before raw pointers, lambdas come before functors, and *std::array* comes before C style arrays.

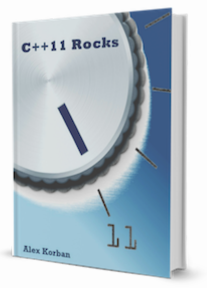
When choosing C++ for a new project, it's a good idea to consider the level of experience of the developers or the time available for learning if C++ is new to them.

I think that it's also good to keep in mind that the complexity of C++ results in a significant disparity in the level of language knowledge between developers, which has to be taken into account in interviews and in providing training. Significant knowledge disparity within a team isn't pleasant for people on either end of the spectrum.

Finally, here are the diagrams side by side for a quick visual comparison:



Discuss:



Keen on mastering C++11/14? I've written books focused on C++11/14 features. You can get a VS2013, Clang or a GCC edition.

C++ Rocks!

This is a resource for developers writing modern C++, with a focus

cpp_rocks

Subscribe via RSS

on the latest revisions of the
language: C++11 and C++14.