To Ruby From C and C++

It's difficult to write a bulleted list describing how your code will be different in Ruby from C or C++ because it's quite a large difference. One reason is that the Ruby runtime does so much for you. Ruby seems about as far as you can get from C's "no hidden mechanism" principle—the whole point of Ruby is to make the human's job easier at the expense of making the runtime shoulder more of the work. Unless or until you profile your code for optimization, you don't need to care one whit about "keeping your compiler happy" when using Ruby.

That said, for one thing, you can expect your Ruby code to execute much more slowly than "equivalent" C or C++ code. At the same time, your head will spin at how rapidly you can get a Ruby program up and running, as well as at how few lines of code it will take to write it. Ruby is much much simpler than C++—it will spoil you rotten.

Ruby is dynamically typed, rather than statically typed—the runtime does as much as possible at run-time. For example, you don't need to know what modules your Ruby program will "link to" (that is, load and use) or what methods it will call ahead of time.

Happily, it turns out that Ruby and C have a healthy symbiotic relationship. Ruby supports so-called "extension modules". These are modules that you can use from your Ruby programs (and which, from the outside, will look and act just like any other Ruby module), but which are written in C. In this way, you can compartmentalize the performance-critical parts of your Ruby software, and smelt those down to pure C.

And, of course, Ruby itself is written in C.

Similarities with C

As with C, in Ruby,…

- You may program procedurally if you like (but it will still be object-oriented behind the scenes).
- Most of the operators are the same (including the compound assignment and also bitwise operators). Though, Ruby doesn't have `++` or `--`.
- You've got `__FILE__` and `__LINE__`.
- You can also have constants, though there's no special `const` keyword. Const-ness is enforced by a naming convention instead— names starting with a capital letter are for constants.
- Strings go in double-quotes.
- Strings are mutable.
- Just like man pages, you can read most docs in your terminal window—though using the `ri` command.
- You've got the same sort of command-line debugger available.

Similarities with C++

As with C++, in Ruby,…

- You've got mostly the same operators (even `::`). `<<` is often used for appending elements to a list. One note though: with Ruby you never use `->`—it's always just `.`.
- `public`, `private`, and `protected` do similar jobs.
- Inheritance syntax is still only one character, but it's `<` instead of `:`.
- You may put your code into "modules", similar to how `namespace` in C++ is used.
- Exceptions work in a similar manner, though the keyword names have been changed to protect the innocent.

Differences from C

Unlike C, in Ruby,…

- Objects are strongly typed (and variable names themselves have no type at all).
- There's no macros or preprocessor. No casts. No pointers (nor pointer arithmetic). No typedefs, sizeof, nor enums.
- There are no header files. You just define your functions (usually referred to as "methods") and classes in the main source code files.
- There's no `#define`. Just use constants instead.
- As of Ruby 1.8, code is interpreted at run-time rather than compiled to any sort of machine- or byte-code.
- All variables live on the heap. Further, you don't need to free them yourself—the garbage collector takes care of that.
- Arguments to methods (i.e. functions) are passed by reference, not by value.
- It's `require 'foo'` instead of `#include <foo>` or `#include "foo"`.
- You cannot drop down to assembly.
- There's no semicolons ending lines.
- You go without parentheses for `if` and `while` condition expressions.
- Parentheses for method (i.e. function) calls are often optional.
- You don't usually use braces—just end multi-line constructs (like `while` loops) with an `end` keyword.
- The `do` keyword is for so-called "blocks". There's no "do statement" like in C.
- The term "block" means something different. It's for a block of code that you associate with a method call so the method body can call out to the block while it executes.
- There are no variable declarations. You just assign to new names on-the-fly when you need them.
- When tested for truth, only `false` and `nil` evaluate to a false value. Everything else is true (including `0`, `0.0`, and `"0"`).
- There is no `char`—they are just 1-letter strings.
- Strings don't end with a null byte.
- Array literals go in brackets instead of braces.
- Arrays just automatically get bigger when you stuff more elements into them.
- If you add two arrays, you get back a new and bigger array (of course, allocated on the heap) instead of doing pointer arithmetic.
- More often than not, everything is an expression (that is, things like `while` statements actually evaluate to an rvalue).

Differences from C++

Unlike C++, in Ruby,…

- There's no explicit references. That is, in Ruby, every variable is just an automatically dereferenced name for some object.
- Objects are strongly but *dynamically* typed. The runtime discovers *at runtime* if that method call actually works.
- The "constructor" is called `initialize` instead of the class name.
- All methods are always virtual.
- "Class" (static) variable names always begin with `@@` (as in `@@total_widgets`).
- You don't directly access member variables—all access to public member variables (known in Ruby as attributes) is via methods.
- It's `self` instead of `this`.
- Some methods end in a '?' or a '!'. It's actually part of the method name.
- There's no multiple inheritance per se. Though Ruby has "mixins" (i.e. you can "inherit" all instance methods of a module).
- There are some enforced case-conventions (ex. class names start with a capital letter, variables start with a lowercase letter).
- Parentheses for method calls are usually optional.
- You can re-open a class anytime and add more methods.
- There's no need of C++ templates (since you can assign any kind of object to a given variable, and types get figured out at runtime anyway). No casting either.
- Iteration is done a bit differently. In Ruby, you don't use a separate iterator object (like `vector<T>::const_iterator iter`) but instead your objects may mixin the `Enumerator` module and just make a method call like `my_obj.each`.

- There's only two container types: `Array` and `Hash`.
- There's no type conversions. With Ruby though, you'll probably find that they aren't necessary.
- Multithreading is built-in, but as of Ruby 1.8 they are "green threads" (implemented only within the interpreter) as opposed to native threads.
- A unit testing lib comes standard with Ruby.
