Extracted from:

Programming Crystal

Create High-Performance, Safe, Concurrent Apps

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Programming Crystal

Create High-Performance, Safe, Concurrent Apps

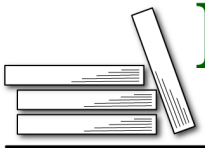
Ivo Balbaert
Simon St. Laurent
edited by Andrea Stewart

Programming Crystal

Create High-Performance, Safe, Concurrent Apps

Ivo Balbaert
Simon St. Laurent

The Pragmatic Bookshelf
Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Susan Conant
Development Editor: Andrea Stewart
Copy Editor: Nancy Rapoport
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-286-2
Book version: P1.0—February 2019

In this chapter, you'll work through a Crystal tutorial. You'll learn to create and manipulate simple kinds of data and data structures. You'll write code that decides what to do based on the data it receives, and you'll learn how to structure that code into methods, classes, modules, and fibers. This chapter will give you the foundation you need to do simple things in Crystal, and later chapters will provide much more detail about larger possibilities.

If you're a Rubyist, Crystal should make you feel right at home. You can probably skim through, focusing especially on differences from Ruby. In fact, in *Organizing Code in Classes and Modules*, on page ?, you'll port some Ruby code step-by-step to Crystal. You'll soon realize that with Crystal, it's all about the messages you get at compile-time, while with Ruby, everything happens at runtime.

If you're not a Ruby developer, the syntax may seem strange at first, but in time, you'll see how concise and intuitive it is.

Then, in Part II, we'll take a closer look at some important details and gain more insight into why Crystal works the way it does.

Start up your Crystal Playground[1] environment to follow along.

Using Basic Variables and Types

Programming is largely about moving data from place to place, and you'll need containers for that data. Crystal's variables can store different kinds of data. As in most languages, you assign them with the = sign.

1. https://play.crystal-lang.org/

foundations/variables.cr
```
name = "Diamond"
element = 'C'
hardness = 10
```

Crystal assignment does more than just put a value into the variable, however. The compiler infers the type of a variable from the value(s) assigned. You can see the variable types the compiler inferred—through type reflection—by testing with the typeof expression.

foundations/variables.cr
```
puts typeof(name)      # => String
puts typeof(element)   # => Char
puts typeof(hardness)  # => Int32
```

Crystal interpreted the name variable as a String because you used double quotes. It interpreted element as a Char (a single character value) because you used single quotes. It interpreted hardness as an integer, specifically a 32-bit integer, because 10 is a short number with no decimal point. (typeof is a great diagnostic tool, but if you're building it into production code, you're working against Crystal's emphasis on managing type details during compilation.)

Most of the time, you can trust Crystal to infer the variable type you want. If you want to be more explicit about it, you can tell the compiler what you want, and the compiler will enforce it for you.

foundations/variables.cr
```
hardness : Int32
hardness = 10
puts typeof(hardness) # => Int32
hardness = "20"  # => Error... type must be Int32, not (Int32 | String)
```

As the first line of that shows, you can declare a variable and its type before assigning it. The Crystal compiler will then use that type in deciding whether later assignments are appropriate. But you can't use that variable for anything before assigning a value to it. (Unless you explicitly define the type of a variable, Crystal will let you assign values of different types to it, and the compiler will keep up if possible.)

You can also declare and assign in the same line, like hardness : Int32 = 10. (Int32 is a signed 32-bit integer, and the spaces around the colon are required.) Remember, however, that you can let Crystal infer variable types most of the time, and you don't need to fill your code with type declarations.

When it's convenient, you can also assign values to multiple variables at one time.

foundations/variables.cr
```
name, element, hardness = "Diamond", 'C', 10
```

If you'd prefer, you can also put multiple statements on one line if you separate them with a semicolon.

foundations/variables.cr
```
name = "Diamond"; element = 'C'; hardness = 10
```

Semicolons enable you to put multiple statements of any kind on a line, not just variable assignments.

If you're in a hurry, you can also switch or swap the values of two variables in one line:

crystal_new/variables.cr
```
# swap
n = 41
m = 42
n, m = m, n
n # => 42
m # => 41
```

(Behind the scenes, Crystal's compiler creates a temporary variable to make this work.)

You may want to create named values that take only one value, called constants. Normal variables, which can change value, start with lowercase letters and use underscores for spaces, called snake or underscore case. Constants, which can't change value, start with uppercase letters, and they're traditionally written in all capitals with underscores for spaces. If you try to set a value for them after they've been set, the Crystal compiler will protest.

foundations/variables.cr
```
DIAMOND_HARDNESS = 10
DIAMOND_HARDNESS = 20  # => already initialized constant DIAMOND_HARDNESS
```

Crystal also includes some starkly limited types that are useful for logical operations. Boolean variables can accept the values of true and false, while nil (of type Nil) explicitly means there is no value.

Not Global

Ruby supports global variables whose names start with $. Crystal's variables are always locally scoped, and it has no global variables.

Variable Operations

Now that you have containers to hold values, you'll want to do things with those values. Crystal offers the traditional mathematical operations, with a few variations depending on whether floats or integers are being processed. (You can mix floats and integers as well.)

foundations/operations.cr

```
d = 10 + 2   # => 12
e = 36 - 12  # => 24
f = 7 * 8    # => 56
g = 37 / 8   # => 4 (integer division)
h = 37 % 8   # => 5 (integers remainder / mod)
i = 36.0 / 8 # => 4.5 (float, or use fdiv function)
```

Strings support the + operator for concatenation that is found in many programming languages.

foundations/operations.cr

```
"this" + "is" + "a" + "test" # => thisisatest
"this " + "is " + "a " + "test" # => this is a test
```

Concatenation works, but it's a clumsy tool. The first item has to be a string for it to work at all, and it'll break if you try to add a number, Ruby style. Fortunately, Crystal lets you assemble strings more naturally with interpolation.

foundations/operations.cr

```
name = "Diamond"
hardness = 10
"The hardness of #{name} is #{hardness}." # => The hardness of Diamond is 10.
```

Crystal evaluates the value of any #{expression} syntax, converts it to a string if necessary, and combines it with the rest of the string.

Underneath these operators is a key truth of Crystal: all of these values are objects. Crystal maps common operator syntax to methods, making it easier for you to write readable code. The type corresponds to a class, which means that all of these values have methods you can use. For example, size is a method on String objects, which returns the number of characters as an Int32.

foundations/operations.cr

```
name = "Diamond"
hardness = 10
name.size     # => 7
hardness.size # => compilation error - undefined method 'size' for Int32
```

But size isn't a method for Int32, so the compiler will give you an "undefined method 'size' on Int32" error.

Your Turn 1

➤ Try out and explain the output of the following statements in Crystal Playground:

```
12 + 12
"12 + 12"
"12" + "12"
"I" * 5
'12' + '12'
5 * "I"
"12" + 12
"2" * "5"
```

Structuring Data with Container Types

Simple variables provide a solid foundation, but as your programs grow, you won't want to keep juggling hundreds of variable names in your head. Crystal provides a number of collection types that let you store a lot of information in structured ways. Arrays and hashes will get you started, and tuples and sets will appear later in the book.

Using Arrays

Sometimes you need to create a list of values, kept in order, and accessible by position. Because this is Crystal, let's collect some minerals. (All of these mineral names are real, and there are lots of them!). The simplest way is to use an array:

foundations/compound_types_arrays.cr

```
minerals = ["alunite", "chromium", "vlasovite"]
typeof(minerals) # => Array(String)

# or, to use a different notation
minerals2 = %w(alunite chromium vlasovite)
typeof(minerals2) # => Array(String)
```

Crystal tells you that `minerals` isn't just an array; it's an array of `String`. Its type, `Array(String)`, also contains the type of the items it contains.

You can add minerals easily with the `<<` operator. The `size` method tells you the number of items it contains:

foundations/compound_types_arrays.cr

```
minerals << "wagnerite"
minerals << "muscovite"
minerals
# => ["alunite", "chromium", "vlasovite", "wagnerite", "muscovite"]
minerals.size # => 5
```

Crystal checks the contents: adding something of another item type isn't allowed. Try adding a number:

foundations/compound_types_arrays.cr
```
minerals << 42
# => Error: no overload matches 'Array(String)#<<' with type Int32
```

You'll get an error while compiling. You'll also get an error if you try to start with an empty array:

foundations/compound_types_arrays.cr
```
precious_minerals = []
# => Error: for empty arrays use '[] of ElementType'
```

This is because the compiler doesn't have enough information to infer its type, and so it can't allocate memory for the array. This is a sharp contrast to Ruby practice, where the type is figured out at runtime. You can create an empty array, but you need to specify a type, either with the [] notation or by creating an object of class Array with new:

foundations/compound_types_arrays.cr
```
precious_minerals = [] of String
precious_minerals2 = Array(String).new
```

As you'd expect, you can read items by index, the position of the item in the array:

foundations/compound_types_arrays.cr
```
minerals[0]  # => "alunite"
minerals[3]  # => "wagnerite"
minerals[-2] # => "wagnerite"
# negative indices count from the end, which is -1
```

You can read subarrays, contiguous sections of arrays, in two different ways. You can give a start index and a size, or use an index range:

foundations/compound_types_arrays.cr
```
minerals[2, 3] # => ["vlasovite", "wagnerite", "muscovite"]
minerals[2..4] # => ["vlasovite", "wagnerite", "muscovite"]
```

What if you use a wrong index? The first item in a Crystal array is always at index 0, and the last at index size - 1. If you try to retrieve something outside of that range, you'll get an error, unlike most Crystal errors, at *runtime*:

foundations/compound_types_arrays.cr
```
minerals[7] # => Runtime error: Index out of bounds (IndexError)
```

If your program logic requires that you sometimes look for keys that don't exist, you can avoid this error. Use the []? method, which returns nil instead of crashing the program:

foundations/compound_types_arrays.cr
```
minerals[7]? # => nil
```

You saw in Chapter 1 that the compiler prevents the use of nil when disaster lurks around the corner. You will soon see how to deal with this in an elegant way.

What if you try to put items of different types in your array?

foundations/compound_types_arrays.cr
```
pseudo_minerals = ["alunite", 'C', 42]
```

This works, but the resulting type of this array is peculiar:

foundations/compound_types_arrays.cr
```
typeof(pseudo_minerals) # => Array(Char | Int32 | String)
```

The compiler infers that the item type is either Char, Int32, or String. In other words, any given item is of the *union type* Char | Int32 | String. (If you want to structure a variable so that it contains specific types at specific positions, you should explore tuples.)

Union types are a powerful feature of Crystal: an expression can have a set of multiple types at compile time, and the compiler meticulously checks that all method calls are allowed for all of these types. You'll encounter more examples of union types and how to use them later in the book.

The includes? method lets you check that a certain item exists in an array.

```
arr = [56, 123, 5, 42, 108]
arr.includes? 42  # => true
```

Need to remove the start or end item? shift and pop can help.

```
p arr.shift # => 56
p arr       # => [123, 5, 42, 108]
p arr.pop   # => 108
p arr       # => [123, 5, 42]
```

If you want to loop through every value in an array, it's best if you do it with the each method or one of its variants.

```
arr.each do |i|
  puts i
end
```

The API for class Array[2] describes many more useful methods. Also, arrays can add and delete items because they are stored in heap memory, and have no

2. https://crystal-lang.org/api/master/Array.html

fixed size. If you need raw performance, use a StaticArray, which is a fixed-size array that is allocated on the stack during compilation.

Displaying Arrays

Want to show an array? Here are some quick options:

```
arr = [1, 'a', "Crystal", 3.14]
print arr               # [1, 'a', "Crystal", 3.14] (no newline)
puts arr                # [1, 'a', "Crystal", 3.14]
p arr                   # [1, 'a', "Crystal", 3.14]
pp arr                  # [1, 'a', "Crystal", 3.14]
p arr.inspect           # "[1, 'a', \"Crystal\", 3.14]"
printf("%s", arr[1])    # a (no newline)
p sprintf("%s", arr[1]) # "a"
```

pp and inspect are useful for debugging. printf and sprintf accept a format string like in C, the latter returning a String.

Your Turn 2

➤ Deleting by value: Most of the time, when you work with arrays, you'll want to manipulate their content based on the positions of items. However, Crystal also lets you manipulate their content based on the values of items. Explore the Crystal API documentation and figure out how to go from ["alunite", "chromium", "vlasovite"] to ["alunite", "vlasovite"] without referencing the positions of the values.

```
minerals = ["alunite", "chromium", "vlasovite"]
minerals.delete("chromium")

p minerals  #=> ["alunite", "vlasovite"]
```

Using Hashes

Arrays are great if you want to retrieve information based on its location in a set, but sometimes you want to retrieve information based on a key value instead. Hashes make that easy.

Let's build a collection that contains minerals and their hardness property using the Mohs Hardness Scale. Given a mineral name, you need to quickly find its hardness. For this, a hash (sometimes called a map or dictionary) is ideal:

foundations/compound_types_hashes.cr

```
mohs = {
  "talc"     => 1,
  "calcite"  => 3,
  "apatite"  => 5,
  "corundum" => 9,
}
typeof(mohs) # => Hash(String, Int32)
```

Its type, Hash(String, Int32), is based on the types of key (String) and value (Int32). You can quickly get the value for a given key using key indexing:

foundations/compound_types_hashes.cr
```
mohs["apatite"] # => 5
```

What if the key doesn't exist, like "gold"? Then, as you saw with arrays, you'll get a runtime error:

foundations/compound_types_hashes.cr
```
mohs["gold"]
# => Runtime error: Missing hash key: "gold" (KeyError)
```

As you saw with arrays, if your logic needs to handle a situation where the key doesn't exist, it's safer to use the []? variant, which returns nil:

foundations/compound_types_hashes.cr
```
mohs["gold"]? # => nil
```

Or, still better, check the existence of the key with has_key?:

foundations/compound_types_hashes.cr
```
mohs.has_key? "gold" # => false
```

Adding a new key-value pair, or changing an existing pair, is easy. You'll use the index notation from arrays, except that you now use the key instead of the index:

foundations/compound_types_hashes.cr
```
mohs["diamond"] = 9  # adding key
mohs["diamond"] = 10 # changing value
mohs
# => {"talc" => 1, "calcite" => 3, "apatite" => 5,
#     "corundum" => 9, "diamond" => 10}
mohs.size # => 5
```

Notice that the size of the hash has increased from 4 to 5. What happens when you add a (key, value) combination where the type of key or value differs from the original items?

foundations/compound_types_hashes.cr
```
mohs['C'] = 4.5 # Error: no overload matches
# 'Hash(String, Int32)#[]=' with types Char, Float64
```

Again, you'll get an error at compile-time: Crystal statically controls your types!

What if you want to start off with an empty hash?

foundations/compound_types_hashes.cr
```
mohs = {}    # Error: Syntax error: for empty hashes use
# '{} of KeyType => ValueType'
```

This doesn't work. Just as with arrays, you need to specify the types again:

foundations/compound_types_hashes.cr

```
mohs = {} of String => Int32 # {}
mohs = Hash(String, Int32).new
```

As you can guess by now, hashes inherit all their methods from the Hash class, which you can find in the API docs.[3]

Your Turn 3

➤ Is that hash empty? Crystal will let you create an empty hash as long as you specify types for both the values and the keys. But empty hashes can create errors and nils when you aren't expecting them. Explore the API docs and find the ways to test for empty hashes and manipulate hashes safely, even when they might be empty.

```
mohs = {
  "talc" => 1,
  "calcite" => 3,
  "apatite" => 5,
  "corundum" => 9
} of String => Int32

p mohs.empty?  => false
```

3. https://crystal-lang.org/api/master/Hash.html