Andrew Shitov

A Language a Day

A brief introduction to 21 programming languages,

alphabetically sorted

DeepText — 2024

A LANGUAGE A DAY

A brief introduction to 21 programming languages

This book provides a concise overview of 21 different programming languages. Each language is introduced using the same approach: solving several programming problems to showcase its features and capabilities.

Code on GitHub: github.com/ash/a-language-a-day

The background image on the cover was generated using AI.
All the programs and text in this book are created by a human.

ISBN: 9789082156874, 9798344731834

1st edition: October 28, 2024

Published by DeepText, Amsterdam
andrewshitov.com/books

Languages

Preface

Programming languages are fascinating. As a software developer, you probably have your own favourites, regardless of the technical nature of the language. You might love its syntax, or its weirdness, or simply follow the trend by learning the most popular language or lucrative one. But beyond all that, there is a certain beauty in how programming languages are crafted. They reflect the vision of their creators, whether it's a dream of building a masterpiece or a design for industrial-scale applications. No matter the intention, every language offers a unique perspective worth exploring.

Once you start experimenting with a few simple test programs, you begin appreciating the language even more. The deeper you dive, the more you understand the design ideas behind the language.

This book is the result of my personal Advent Calendar project. With a few exceptions, I explored a different language every day—some I had experience with, some I only knew by name, and others were completely new to me.

Over the past decade, numerous programming languages emerged. Some have risen to prominence, while others remain more niche. Backed by large IT companies or driven by passionate individuals, they represent a diverse range of ideas.

I am sure you can name a dozen languages you are aware of but never tried. The same goes for me. I've heard of many new languages but have either never used them or only experimented with a simple *Hello, World!* example.

Each chapter covers the essentials of a different programming language. To make the content more consistent and comparable, I use the same structure for each language, focusing on the following mini projects:

1. *Creating a 'Hello, World!' program.*
2. *Implementing a Factorial function using recursion or a functional-style approach.*
3. *Creating a polymorphic array of objects (a 'zoo' of cats and dogs) and calling methods on them.*
4. *Implementing the Sleep Sort algorithm—while impractical for real-word use, it's a playful demonstration of language's concurrency capabilities.*

Each language description follows—where applicable—this pattern:

1. *Installing a command-line compiler and running a program.*
2. *Creating and using variables.*
3. *Defining and using functions.*
4. *Exploring object-oriented features.*
5. *Handling exception.*
6. *Introducing basic concurrency and parallelism.*

You can find all the code examples in this book on GitHub: github.com/ash/a-language-a-day.

The languages are listed alphabetically for easy reference.

If you wish to contact the author, feel free to write to mail@andrewshitov.com.

C++

C++ feels like a new language.

— Bjarne Stroustrup

You may reasonably ask what's up with C++? It first appeared about 40 years ago. But in 2011, the new era of C++ began. If you haven't followed its development over the last 5-10 years, you may be surprised by its significant advancements.

Facts about the language

- A compiled language
- Based on C but goes far beyond
- Designed to have zero overhead (no extra code generated for the features that are not directly used)
- Provides strong object-oriented support
- Released in 1985, "reinvented" in 2011
- Website: isocpp.org

Installing and running C++

If you are working in a Linux environment, you likely already have the GCC compiler installed. If not, you can install it using your package manager or from source. Mac users can either download the full XCode or just the command-line tools, XCode Developer Tools.

Although there are several other compilers available, the Gnu's GCC C++ compiler will be used in this chapter:

```
$ g++ hello-world.cpp -ohello-world
$ ./hello-world
```

To ensure you're using the correct standard version, specify it in with the `-std` parameter. For example:

```
$ g++ -std=c++20 auto.cpp
```

You can use `c++11`, `c++14`, `c++17`, or `c++20` values for the corresponding standard. (Use `c++2a` with the earlier versions of GCC.) The most recent language standard is C++23, which has the corresponding experimental support through the options `c++2b` or `gnu++2b`.

Hello, World!

Without further ado, here's the code:

C++/hello-world.cpp

```
#include <iostream>

int main() {
    std::cout << "Hello, World!\n";
}
```

The program prints the expected greeting:

```
Hello, World!
```

There are two key points to note here (although these features have been available for a long time). First, the header file no longer requires the `.h` extension. Second, it is now acceptable to omit returning a value from the `main` function.

If you prefer not to use the overloaded operator `<<` for printing, you can use functions instead:

C++/hello-world-print.cpp

```
#include <iostream>

int main() {
    std::printf("Hello, World!\n");
}
```

The output is the same.

Variables

The compiler can now infer the type of the variable automatically. Simply use the `auto` keyword instead of the explicit variable type.

C++/auto.cpp

```
#include <iostream>

int main() {
    auto name = "John";
    std::cout << "Hello, " << name << "!\n";
}
```

It prints a concatenated string:

```
Hello, John!
```

Functions

Since C++ 14, type deduction is also available for the return values of functions:

C++/auto2.cpp

```
#include <iostream>

auto sqr(int x) {
    return x * x;
}

int main() {
    std::cout << sqr(5) << std::endl;
}
```

To compile this program, use the `-std=c++14` option or a higher version.

The program prints the square to the console:

```
25
```

Lambdas

The new C++ introduces syntax for creating lambda functions. Consider the following example, where we pass our custom predicate to the `sort` function from the standard library. Instead of creating a separate function, you can define the action directly where it is used.

C++/lambda.cpp

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main() {
    vector<int> data {10, 4, 2, 6, 5, 7, 1, 3};

    sort(data.begin(), data.end(),
         [](int a, int b){return b < a;}
    );

    for (auto x : data) cout << x << endl;
}
```

The code in bold is an anonymous two-argument function that returns a Boolean value. This is what the third optional parameter that `sort` expects. In practice, you can also use the `std::greater` function. The output is shown below:

```
10
7
6
5
4
3
2
1
```

Containers

When working with the standard library, particularly with its containers, a lot of typing was previously necessary. To illustrate this, compare the following two versions of a program that prints three strings from a vector.

C++/vector-oldstyle.cpp

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<string> v;
    v.push_back("alpha");
    v.push_back("beta");
    v.push_back("gamma");

    for (vector<string>::const_iterator i = v.begin();
         i != v.end(); i++)
        cout << *i << "\n";
}
```

In the first version, you had to manually populate the vector with data use an explicit iterator, whose type is a long string of characters, which is, by the way, already shortened by using the namespace in this program. Without that, it would be `std::vector<std::string>::const_iterator`.

Today, you can do it much more easily:

C++/vector-newstyle.cpp

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<string> v {"alpha", "beta", "gamma"};

    for (auto s : v) cout << s << "\n";
}
```

Modern C++ allows *initializer lists*, so you can set the initial values to a container in a way similar to arrays. The `for` loop here is the so-called range-for loop. Along with `auto`, it simplifies the process by handling the container's type and contents automatically, while still maintaining full static typing.

Both programs shown above produce identical output:

```
alpha
beta
gamma
```

Classes

C++'s has a well-known object-oriented model among developers. This chapter focuses on the new versions of C++, so let us dive straight into the code.

A Polymorphic example

Here is a zoo with both cats and dogs. The goal of this program is to demonstrate how to use the same name in a list of objects of different types, and still correctly select the method belonging to the proper class definition.

The program defines three classes: `Animal` as the base class, and `Cat` and `Dog` as the two child classes. Then, a vector is filled with both cats and dog. The vector does not have any prior information about whether its elements are dogs or cats, because its elements are of the `Animal*` type.

C++/zoo.cpp

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

class Animal {
public:
    virtual string info() = 0;
};

class Cat: public Animal {
public:
    virtual string info() {
        return "Cat";
    }
};

class Dog: public Animal {
public:
    virtual string info() {
        return "Dog";
    }
};

int main() {
    vector<Animal*> zoo {
        new Cat(), new Cat(), new Dog(), new Dog()
    };
```

```
    for (auto x : zoo) cout << x->info() << "\n";
}
```

The `Animal` class is a base class for both `Dog` and `Cat` classes, featuring a pure virtual method `info()` that returns a `string`. The virtual nature of the method is indicated by the `virtual` keyword, and `=0` denotes its *pure* virtual status.

The program calls the `info()` methods from the correct child class, resulting in the following output:

```
Cat
Cat
Dog
Dog
```

Concurrency and threads

Modern C++ supports threads directly via the standard library, making it available on multiple platforms. Syntactically, it is a standard C++, but it may appear as syntax sugar to conceal multitasking management behind a few keywords.

Sleep Sort

The program below is a full implementation of the Sleep Sort algorithm using threads. It creates a thread for each number in the `databox` vector. The threads execute the function to wait for a delay proportional to the number before printing the value. Different approaches can be experimented with when choosing the relationship between the number and the delay. In this program, the number to be sorted represents the number of milliseconds in the delay.

C++/sleep-sort.cpp

```
#include <iostream>
#include <vector>
#include <thread>

using namespace std;

void sort_me(int n) {
    this_thread::sleep_for(chrono::milliseconds(n));
    cout << n << endl;
}
```

```
int main() {
    vector<int> databox {10, 4, 2, 6, 5, 7, 1, 3};

    vector<thread> tasks;
    for (auto n : databox) tasks.push_back(thread {sort_me, n});
    for (auto&& t : tasks) t.join();
}
```

Note the two range-based `for` loops in the `main` function. The first loop stores a pointer to a new thread in the `tasks` vector. In the second loop, this vector is used to merge all the threads before exiting the program.

Run the program to confirm that the numbers are indeed sorted.

```
1
2
3
4
5
6
7
10
```

Thread-safety

The Sleep Sort code shown above has a small problem: its output is not thread-safe. If smaller delays (e. g. microseconds) are used and large numbers are sorted, incorrect output may be produced, including misplaced newlines and digits. To prevent this issues, a mutex should be added:

C++/sleep-sort-mutex.cpp

```
#include <mutex>

. . .

void sort_me(int n) {
    static mutex mu;


    this_thread::sleep_for(chrono::milliseconds(n));

    unique_lock<mutex> lock {mu};
    cout << n << endl; // thread-safe now
}
```

Using the `unique_lock` template simplifies locking and releasing the mutex with a single line of code. The release happens in the destructor of the `lock` object, which is called at the end of its scope, which in this case is the `sort_me` function.

Get more!

C++ is a great language and I hope that if you used to use it a decade ago, you may reconsider using it again. Use the following resources to learn more about the modern versions of the language.

- *Standard* C++ https://isocpp.org/
- C++ *reference* https://www.cppreference.com/
- *A Tour of* C++, 2nd edition, the book by B. Stroustrup

Factor

Facts about the language

Some facts about the Factor programming language:

- A *concatenative* stack-based language
- Dynamically typed
- Appeared in 2003
- Website: factorcode.org

Installing and running Factor

You can compile Factor from sources or download an image for your operating system. This method is quite unusual: it provides you with a mounted image with a binary file and all the necessary files. There is a visual IDE, Factor.app (for Mac), but let me use the command-line tool. For simplicity, you can export the `PATH`:

```
$ export PATH=$PATH:/Volumes/factor/factor/
```

Having this done, run a Factor program as:

```
$ factor helloworld.factor
```

Hello, World!

Here is the minimum program in Factor:

Factor/helloworld.exs

```
USE: io

"Hello, World!" print
```

The program puts `"Hello, World!"` on the stack and applies the `print` word to it. To use `print`, you need to load the *vocabulary* `io` first.

The output looks as follows:

```
Hello, World!
```

Output

There are a few ways to print to the console that we'll use in the rest of this chapter. Here they are shown in a small program:

Factor/prints.exs

```
USING: io present prettyprint ;

"factor" .
"factor" print

42 .
42 present print
42 number>string print
```

The `.` is probably the easiest way to print values, both numbers and strings. The difference is that `.` additionally adds the quotation marks in the output:

```
"factor"
factor
```

When you print numbers with `print`, you need to convert them to a string first by applying `present`. Alternatively, use the `number>string` word to convert the type.

```
42
42
42
```

Stack

In the above Factor program, we can see *words* and *literals*. A literal, `"Hello, World!"`, puts itself on the stack. A word, `print`, performs an action: it takes the top object from the stack and prints it.

For example, push two strings to the stack and call `print` twice:

Factor/printprint.exs

```
USE: io
"Hello," "World!" print print
```

This program prints the two lines with the two given strings:

```
World!
Hello,
```

You should keep the same considerations in mind when dealing with arithmetic tasks. This is how you print the sum of two numbers:

Factor/math.exs

```
USING: math prettyprint ;

40 2 + .
```

The two numbers are put on the stack, then the `+` word is applied, which takes the two values from stack, computes the sum, and puts it back on the stack. The `print` word then takes it from the stack and prints it. The word `present` is needed here to get a string representation of a `fixnum` data type.

Be careful when the operation is not commutative:

```
40 2 - .
```

Although the `2` is on the top of the stack, it is the second operand for the `-` word.

Comments in Factor start with `!`.

The program prints the results:

```
40
38
```

Variables

The stack and variables are kind of mutually exclusive. Nevertheless, there exist lexical, dynamic, and global variables. For the sake of simplicity, let us only look at lexical variables. We need to define the scope, and you do it with the `[let … ]` construct.

Factor/variable.exs

```
USING: locals prettyprint ;

[let 42 :> answer
    answer .
]
```

The program prints the value assigned to the variable:

```
42
```

Note that there should be no space between the opening bracket and `let`. The whole `[let` is a single word and you can't split it. Words are separated by spaces, and that is the reason, for example, why you need to put a space before the semicolon after `USING:`.

If you need more than a single variable, just list them all:

Factor/variable2.exs

```
USING: math prettyprint ;

[let 100 :> a 200 :> b
    a b + .
]
```

This program will print the sum of 100 and 200:

```
300
```

Defining words

You can think of a word as a function that takes its arguments from the stack and puts the result back. To define a new word, use the `::` syntax. You also need to declare the *stack effect*, i.e. the number of elements to be taken from and to be placed on the stack.

Factor/word.exs

```
USING: io locals sequences ;

IN: myprogram

:: greet ( name -- phrase )
    "Hello, " name append
```

```
    "!" append
    ;

"John" greet print
```

Here, we define the `greet` word, and it has to be in a vocabulary. For this, the `IN:` word is used. Here, we define a vocabulary named `myprogram`.

The stack effect `( name -- phrase )` informs that the word takes one element from the stack and puts one element onto it. The word's body concatenates a few strings and leaves the result on the stack. Later, we can use the newly defined word and apply it to the `"John"` literal. The program will create a string and print it:

```
Hello, John!
```

A Factorial example

Let us define factorial as a word that takes a value from the stack and puts the result back.

Factor/factorial.exs

```
USING: kernel math prettyprint ;
IN: myprogram

:: factorial ( n -- f )
    n 2 < [ 1 ] [ n n 1 - factorial * ] if
    ;

1 factorial . ! 1
5 factorial . ! 120
7 factorial . ! 5040
```

The `factorial` function is recursive, and thus it needs a condition `n 2 <` … `if` to stop iterating. Notice how the product of $n * (n-1)!$ is spelled out: `n n 1 - factorial *`. Alternatively, we could arrange the operands differently: `n 1 - factorial n *`.

The program prints the desired results:

```
1
120
5040
```

It is possible to create a more compact factorial definition using reduction. Here is the new definition of the factorial function:

Factor/factorial2.exs

```
USING: math prettyprint ranges sequences ;

IN: myprogram

: factorial ( n -- n! )
    [1..b] product
;
```

The word `factorial` is defined with a single colon now. In this case, the arguments are not bound to lexical variables, and the function does not need them. The name of the output variable is not used in either case. Note that it is not restricted to letters only: you can describe the stack effect as `( n -- f )` or `( n -- n! )` whichever you like more.

You can use the new version of the `factorial` function in the same was as earlier:

```
1 factorial . ! 1
5 factorial . ! 120
7 factorial . ! 5040
```

Let us examine what's inside the function. The first word (there are no spaces in it!) is `[1..b]`, which means the range from 1 to the value on the top of the stack with an interval of 1.

Note that `b` in the word is not the name of a variable or a function argument; it is a part of the word. There are other ranges defined in Factor, for example, `[1..b)` to exclude the last value, or `[a..b]` that need two values on the stack and creates a range from a to b including both ends.

For example, the following program computes and prints the product of the numbers from 5 to 10:

Factor/product.exs

```
USING: prettyprint ranges sequences ;

5 10 [a..b] product .
```

The result is:

```
151200
```

Finally, the Factor library already includes the implementation of the `factorial` function, so it is possible to use it directly:

Factor/product.exs

```
USING: math.factorials prettyprint ;

1 factorial . ! 1
5 factorial . ! 120
7 factorial . ! 5040
```

This program will print the same result as the previous program with our own implementation of the `factorial` word.

Object-oriented programming

Classes in Factor are implemented via tuples. In the following example, a tuple definition is created and then a new object is created:

Factor/person.exs

```
USING: io accessors kernel ;

IN: people

TUPLE: person name ;

person new
"John" >>name

name>> print
```

The `accessors` vocabulary creates special words, `>>name` and `name>>` for setting and getting the fields of the `person` tuple.

First, `person new` creates a new object and puts it on the stack. Then, we set the name by applying `>>name` to the two top stack values, which are the tuple itself and the string `"John"`. At this moment, `person` is still on the stack.

To print the name, `name>>` is called, which removes the person from the stack and puts its `name` there.

The program prints the person's name:

```
John
```

How do you add another field to the class? The tuple definition should be changed to include it:

```
TUPLE: person name age ;
```

Setting the data is also straightforward:

```
person new
"John" >>name
22 >>age
```

Reading two fields is a bit trickier. As getting the name via `name>>` removes the object from the stack, you need to duplicate it first so that you have its copy (a reference) when you are reading its `age`:

```
dup
name>> print
age>> present print
```

The whole program now looks like this:

Factor/person2.exs

```
USING: io accessors kernel present ;

IN: people

TUPLE: person name age ;

person new
    "John" >>name
    22 >>age

dup
name>> print
age>> present print
```

It prints both the name and the age:

```
John
22
```

To reinforce our understanding, let us create a program that creates two `person` objects, saves them in variables, and then prints a summary for both. Examine the following program.

Factor/person3.exs

```
USING: accessors formatting io kernel math present strings ;

IN: people

TUPLE: person
    { name string }
    { age integer }
;

[let
    person new
        "John" >>name
        22 >>age
    :> p1

    person new
        "Alla" >>name
        20 >>age
    :> p2

    p1 name>> p1 age>> "%s is %d.\n" printf
    p2 name>> p2 age>> "%s is %d.\n" printf
]
```

Here, the tuple `person` contains two fields, and it also specifies their types:

```
TUPLE: person
    { name string }
    { age integer }
;
```

Then, inside the `[let … ]` construct, two variables `p1` and `p2` are created. These are lexical variables, so we must use them before the final closing bracket matching `[let`.

To print the string, the `printf` word is used. To prepare data for it, we need to put three elements onto the stack: the name and age fields on a variable `p1` or `p2`, and the formatting string `"%s is %d.\n"`.

The program now prints the following two strings:

```
John is 22.
Alla is 20.
```

Inheritance

Factor allows us to describe class inheritance. Let us create the zoo program to demonstrate polymorphism.

The main idea is to create a base class `animal` and define two child classes: `dog` and `cat`. In Factor, the following syntax is used:

```
TUPLE: animal name ;
TUPLE: cat < animal ;
TUPLE: dog < animal ;
```

We'll also need a method that returns the kind of animal. Let us define one for each of the subclasses.

As we are going to call the method on the objects in the list where we do not know which class the list items belong to, we need to declare a generic method in the animal class:

```
GENERIC: kind ( animal -- kind )
```

Now it is possible to create individual implementations for the subclasses:

```
M: dog kind ( dog -- kind )
    drop "dog" ;

M: cat kind ( cat -- kind )
    drop "cat" ;
```

The stack effects `( dog -- kind )` and `( cat -- kind )` inform that the `kind` method expects one object on the stack (`cat` or `dog`) and puts a single result on the stack, which is a string in our example. The method does nothing with the input, so we remove it from the stack using `drop`.

Here is the entire program:

Factor/zoo.exs

```
USING: accessors arrays classes.tuple.private formatting
inspector io kernel math present prettyprint sequences ;
```

```
IN: zoo

TUPLE: animal name ;
TUPLE: cat < animal ;
TUPLE: dog < animal ;

GENERIC: kind ( animal -- kind )

M: dog kind ( dog -- kind )
    drop "dog" ;

M: cat kind ( cat -- kind )
    drop "cat" ;

[let
    cat new "Catty" >>name :> a1
    cat new "Kitty" >>name :> a2
    dog new "Doggy" >>name :> a3
    dog new "Diggy" >>name :> a4

    { a1 a2 a3 a4 } :> animals

    animals [
        dup
        name>> swap kind "%s is a %s.\n" printf
    ] each
]
```

After all the class definitions, the program creates four objects of `cat`s and `dog`s, gives them names, and puts them into the `animal` array.

Then there is a loop implemented with `each`. It needs the `animal` array and some code in square brackets, where you pick the required data, arrange it in the correct order, and print using the formatting string.

The `dup` word copies the current object in the loop so that we can access its data twice. First with the `name>>` getter and later by calling the `kind` method. Note that after name>>, there are two elements on the top of the stack: the copy of the object and its name. As `kind` must be called on the object itself, we `swap` the two things on the stack. After calling `kind`, it puts another string to the stack, so the following `printf` consumes these two strings followed by the formatting string.

The program prints:

```
Catty is a cat.
Kitty is a cat.
Doggy is a dog.
Diggy is a dog.
```

Sleep Sort

Before the end of this chapter, let us create a program for the Sleep Sort algorithm. We have already seen a number of elements that we need now. Here is the program:

Factor/zoo.exs

```
USING: calendar concurrency.combinators kernel prettyprint
threads ;

IN: sleep-sort

: sleep-sort ( data -- ) [
    dup seconds sleep .
] parallel-each ;

{ 10 4 2 6 5 7 1 3 } sleep-sort
```

The last line of the program executes the `sleep-sort` function that works on a sequence of the numbers to be sorted.

In the definition of the function, we see a `parallel-each` loop that is similar to what was used in the previous program with animals. But this time, for each of element of the sequence, a new thread is spawned.

Inside the thread (which is the body of the loop), we just `sleep` the given number of seconds and print the value. As both `sleep` and `print` need the number, it is `dup`ped first.

As soon as all the thread complete their work, the program prints the result. All the numbers are printed only after the longest delay is over.

```
1
2
3
4
5
```

```
6
7
10
```

Get more!

The main resources for the Factor language are its own documentation and source code:

- *Factor handbook*
  https://docs.factorcode.org/content/article-handbook.html
- *factor/code repository*
  https://github.com/factor/factor/tree/master/core

Additionally, the following external links may be useful:

- *A panoramic tour of Factor*
  http://andreaferretti.github.io/factor-tutorial/
- *Factor on concatenative.org*
  https://concatenative.org/wiki/view/Factor

andrewshitov.com/books