XièXie Programming Guide

Lukas Hermanns

March 13, 2015

About the Author

My name is Lukas Hermanns (age-group 1990) and I started this project during my studies in 2014. By now I have over 12 years of experience in computer programming, started at the age of 12. I have been writing programs in Basic languages such as QBASIC, PUREBASIC, and BLITZ3D; in high level languages such as C, C++, C#, OBJECTIVE-C, and JAVA; but also in scripting languages such as JAVASCRIPT and PYTHON. I'm actually a preferred programmer in C++ (meanwhile C++11), low level stuff, and graphics programming with shading languages such as GLSL and HLSL.

However, the XIÈXIE programming language is intended to be simple and not tuned for performance. It was originally designed to be used for scripting in video games, but can also be used for general purposes.

Contents

Chapter 1

Introduction

First of all, "xièxie" is the chinese word for "thanks" or "thank you" (see hantrainerpro.com) and is pronounced for instance "Sh-eh sh-eh". To make a long story short,

The design of the XièXie programming language is overall influenced by Java, C++, and Python.

1.1 Motivation

There are many great programming languages out there. Some produce faster code than others, but for that others are simpler and have a better learning curve. Using a new language doesn't mean to give up a previous one, because every language has its own domain. For example, the performance of interpreted languages such as Python is totally sufficient for many applications. We don't need maximal performance for a script which does some socket connection or text processing for instance. But I would not write the compiler and interpreter for Python in a scripted language. There are also many ways to combine several languages: an interpreter could be used for scripting in video games for instance. But the game engine is written in C++. This is how it is done in Unity3D (see www.unity3d.com). They use Mono as compiler and interpreter framework for C#. But the engine itself is written in C++.

Now XièXie is aimed to be used for small scripting purposes, with a gentle learning curve. The great thing about its interpreter is, that it is very tiny and can easily be integrated into existing C or C++ code. This virtual machine only consists of a single code file (written in C99) and can simply be included into any C99 compliant project.

1.2 Why is it called "XièXie"?

Some years, before I started with the development of this compiler, I already had some ideas about a name for it — at this time the compiler was intended to be a cross compiler, i.e. to compile the XièXie code to C++. The first idea I had in mind, was to call it "C+=2" or "C++++" because it should be a more comfortable and easier C++ variant. But this name looked and sounded strange. Another idea was to call it "C power of C" (in mathematical notation C^C). But that still was not what I was looking for. Then I remembered me to the Chinese word "Xièxie" which means "Thanks" in English and it sounded a little similar to "CC" which could be seen as a shortcut for C^C. That's the XièXie compiler's story about its name.

Or to make a long story short: I saw this word on a napkin in a Chinese restaurant and thought to my self: That's it! :-)

Chapter 2

Syntax

The we start out with the syntax.

2.1 Basics

2.1.1 Commentaries

Commentaries are a fundamental part of programming languages and they are nearly identical to those in Java:

```
1   // Single-line comment
2
3   /* Single-line comment */
4
5   /*
6   Multi-line comment
7   */
```

Although they are very similar to the commantaries in Java, nested multi-line comments are allowed as well:

```
1   /*
2   Outer comment
3   /* Nested comment */
4   */
```

2.1.2 Identifiers

Identifiers (for variables, classes, etc.) must only contain alpha-numeric characters and the underscore. They must also begin with a letter or an underscore. But there is a small exception: they must not begin with __xx__, because all identifiers with this prefix are reserved for internal use of the compiler.

Valid identifiers are:

- _name_
- FooBar
- number_of_wheels
- Customer01
- ...

Invalid identifiers are:

- naïve
- Foo.Bar
- number-of-wheels

- 3over2

- . . .

2.1.3 Literals

These are the kinds of literals:

- Boolean Literal: `true`, `false`

- Integer Literal (Binary): e.g. `0b11001`, `0b0000`

- Integer Literal (Octal): e.g. `0o24`, `0o01234567`

- Integer Literal (Decimal): e.g. `3`, `12`, `999`, `1234567890`

- Integer Literal (Hexa-Decimal): e.g. `0xff`, `0x00`, `0xaB29`

- Float Literal: e.g. `0.0`, `3.5`, `12.482`

- String Literal: e.g. `"Foo Bar"`, `"Hello, World"`, `"\n\t"`, `"1st fragment"` `"2nd fragment`

- Verbatim String Literal: e.g. `@"\home\test"`, `@"a ""b"" c"`

2.2 Operators

The most operators as in JAVA or C++ are also available in XIÈXIE:

```
1   /* Arithmetic Operators */
2   a + b   // Addition
3   a - b   // Substration
4   a * b   // Multiplication
5   a / b   // Division
6   a % b   // Modulo
7   a << b  // Left Shift
8   a >> b  // Right Shift
9   -a      // Negate
10
11  /* Bitwise Operators */
12  a & b   // Bitwise AND
13  a | b   // Bitwise OR
14  a ^ b   // Bitwise XOR
15  ~a      // Bitwise NOT
16
17  /* Boolean Operators */
18  not a   // Logic NOT
19  a and b // Logic AND
20  a or b  // Logic OR
21
22  /* Relation Operators */
23  a = b   // Equality
24  a != b  // Inequality
25  a < b   // Less
26  a <= b  // Less Or Equal
27  a > b   // Greater
28  a >= b  // Greater Or Equal
```

As the interested reader may have noticed, the *equality operator* is different to that in the most languages. This is because the *copy assignment* operators is `:=` and not `=`.

```
1   x := 5       // ok, set x to 5
2   a, b, c := 4 // ok, set a, b, and c to 4
3   a := b := 3  // error, b := 3 is not an expression
```

In the above example `a := b := 3` is invalid, because on the right hand side of the first `:=` must be an expression. But per definition in XIÈXIE `b := 3` is not an expression, but a statement! The variable list assignment (`a, b, c := ...`) is a comfort functionality, which is only supported for the copy assignment.

The modify-assign operators are available as well:

```
1  a += b
2  a -= b
3  a *= b
4  a /= b
5  a %= b
6  a <<= b
7  a >>= b
8  a &= b
9  a |= b
10 a ^= b
```

However, they are also not allowed inside another expression.

2.3 Type Denoters

2.3.1 Built-in Types

There are only the following three built-in data types:

```
1  bool  // Boolean type; can be 'true' or 'false'
2  int   // 32-bit signed integral type
3  float // 32-bit floating-point type
```

2.3.2 Objects

Types for class objects (more about classes in chapter 3) are written as follows:

```
1  // Empty string
2  String s
3
4  // List of strings
5  String s1 := "Hello, World", s2 := "Foo", s3 := "Bar"
```

2.3.3 Arrays

The only generic way for lists are the built-in arrays:

```
1  // Declare array objects with initializer lists
2  int[] intArray := { 1, 2, 3 }
3  float[][] floatArrayArray := { { 0.0, 1.5 }, { 3.5, 1.23 } }
4  String[] stringArray := { "a", "b", "c" }
5
6  // Access array elements
7  String s1 := stringArray[0]
8  String s2 := stringArray[intArray[0]]
9  float[] floatArray := floatArrayArray[1]
```

2.3.4 Automatic Type Deduction

Whereas automatic type deduction in C++11 is a very complex language feature, in XɪÈXɪE it can be summarized in this section. There are two keywords for automatic type deduction: var and const. As the name implies var denotes a variable type and const denotes a constant type. The latter type is the only way to define constants in XɪÈXɪE. Here are a few examples:

```
1  var i  := 1     // i is from type 'int'
2  var f  := 3.5   // f is from type 'float'
3  var s  := "."   // s is from type 'pointer of String'
4  var a  := { 5 } // a is from type 'array of int'
5  var aa := {      // aa is from type 'array of array of String'
6    { "test" },
7    { "a", "b" }
8  }
9
10 const ci := 5      // ci is a constnat int with value 5
11 const cj := ci*2   // cj is a constant int with value 10
12 const cf := 3.14   // cf is a constant float
13 const cb := ci > cj // cb is a constant bool with value 'false'
```

Chapter 3

Classes

A XièXie program can only contain of class declarations. And classes can only be defined in the global scope. That means every procedure must be defined inside a class and inner classes are currently not supported.

3.1 Getting Started

To get started, take a look at the following example program which prints the classical phrase "Hello, World!" onto the standard output:

```
// XieXie Hello World Program
import System
class HelloWorld {
    [[entry]]
    static void main() {
        System.out.writeLine("Hello,_World!")
    }
}
```

This merely writes the line "`Hello, World!`" to the standard output. Let's take a closer look at each line.

Line 2 imports the "System.xx" file from the XièXie standard library:

```
import System     // either this ...
import "System.xx" // ... or this
```

If the imported file is in another directory, the string version of `import` is the only choice. This can also be omitted if the file is added to the compilation process. The files for the classes `Object`, `String`, `Array`, and `Intrinsics` are always implicitly imported, because they are 'internal' classes the compiler knows generally. Note that verbatim strings are allowed wherever string literals are allowed, i.e. the following example is valid XièXie code:

```
import "C:\\Program_Files\\Test1.xx" // either this ...
import @"C:\Program_Files\Test1.xx"  // ... or this
```

The `import` keyword is different to that in Java and also different to the `#include` directive in C++. Although it takes a filename as parameter (like C++'s `#include`), it does not *include* the file in place. Whenever an `import` is read by the *parser*, the filename is added to the set of import files. After all source files have been read, which were passed as input to the compiler, all import files will be read next. This will be repeated until no new files are added to the set. Consider this is a *set* of files, i.e. several `import` commands may occur with the same filename, but it will be read only once. This is why the above sample is valid XièXie code. It also means that recursive imports are allowed as well:

```
// File1.xx
import "File2.xx"
```

```
// File2.xx
import "File1.xx"
```

Line 3 declares the class `HelloWorld` which implicitly inherits from the base class `Object`, like it is done in Java:

```
1  class HelloWorld { /* ... */ }
```

To inherit from other classes, just write a colon and the identifier of the base class:

```
1  class SubClass : BaseClass { /* ... */ }
```

There is no multiple inheritance like in C++ or interfaces like in Java!

The next two lines declare the procedure `main`. In line 4, in *attribute* is defined for this procedure. This makes the procedure to the main **entry point**:

```
1  [[entry]]
2  static void main() { /* ... */ }
```

There are several attributes for class-, procedure-, or variable declarations. Currently only two attributes are supported:

```
1   // class A is marked as 'deprecated'
2   [[deprecated]]
3   class A {}
4
5   // class B is marked as 'deprecated' with a hint
6   [[deprecated("hint...")]]
7   class B {
8       // procedure M1 is marked as the main entry point
9       [[entry]]
10      static void M1() {}
11
12      // procedure M2 with return type is marked as an
13      // alternative entry point named "entryAlt1"
14      [[entry("entryAlt1")]]
15      static int M2() { return 0 }
16
17      // procedure M3 with arguments 'args' is marked as
18      // an alternative entry point named "entryAlt2"
19      [[entry("entryAlt2")]]
20      static void M3(String[] args) {}
21  }
```

The last line of code prints the message to the standard output:

```
1  System.out.writeLine("Hello,␣World!")
```

System is a class from the standard XièXie library, out is a *static* member from the type 'Output-Stream', and writeLine is a function which takes a string as input.

Chapter 4

Compiler

4.1 Command Line Tool

The compiler can be used as command line tool. The appropriate program is named **xxc**. Enter `xxc help` in a command line to see the manual pages.

In contrast to most other command line tools, the commands for **xxc** don't have the '-' prefix. Instead the command options use this prefix. Here is an example:

```
1  xxc compile -f FILE1 -f FILE2 -O
```

The above command line uses the `compile` command with the flags '-f' and '-O'.

Chapter 5

Virtual Machine

The XièXie Virtual Machine (XVM) is a separated program (named **xvm**), written in pure C99.

5.1 Executing Programs

To execute (or run) a virtual program, just enter the filename of an ***.xbc** file into the **xvm**:

```
1   xvm HelloWorld.xbc
```