

Table of Contents

0 - Preface

1 - What is Jai ?

- 1.1 Some context and history.
- 1.2 What type of language is Jai?

1B - What is Jai - more in depth

- 1.1 What type of language is Jai?
 - 1.1.1 Priorities
 - 1.1.2 Jai tries to be a better C/C++
- 1.2 Comparisons with other languages
- 1.3 Jai's performance
- 1.4 Some steps in Jai's history
- 1.5 Specific Jai features
- 1.6 Some wrong ideas in software development
- 1.7 Jai community and communication channels
- 1.8 Jai's popularity

2 - Setting up a Jai development environment

- 2.1 Opening up the Jai compiler toolkit
- 2.2 Setting up the Jai compiler
 - 2.2.1 Copying the compiler to its destination folder
 - 2.2.2 Making the jai command system-wide available
 - 2.2.3 Updating Jai and switching versions
 - 2.2.4 Prerequisite for Windows
 - 2.2.5 Windows as development platform
 - 2.2.6 Solution for install problem on Linux distros
 - 2.2.7 Working in WSL on Windows
- 2.3 Editor help for coding Jai
 - 2.3.1 Overview of different editors
 - 2.3.2 Using the Visual Studio Code plugin
 - 2.3.3 How to edit, build and run a Jai program in VS-Code through CodeRunner
- 2.4 The compiler command

2B - Compiler command-line options

3 – Compiling and running your first program

3.1 Some preliminary remarks

3.2 The main entry point

3.3 Compiling our first program

3.3.1 Compile-time

3.3.2 Printing output

3.3.3 Run-time

3.3.4 Running code during compile-time

3.3.5 Some remarks

3.3.6 Errors

3.3.7 Exiting a program

4 – More info about the compiler

4.1 General info

4.2 Internal byte-code interpreter

4.3 Front-end

4.4 Back-ends

4.5 Linking

4.6 Architectures

4.7 Debug and release build

4B_Options for giving code at the command-line

4C_The Preload module

4D_Memory management

4E_What happens when Jai starts up?

5 – Constants, variables, types and operations

5.1 Data, literals and types

5.1.1 Data and types

5.1.2 The primitive types: bool, int, float, string, void

5.1.3 Using print to display a value

5.1.4 type_of()

5.2 Constants

5.2.1 Problem: What if we need the same literal many times in code?

5.2.2 Solution: Constants

5.3 - Variables

5.3.1 - How to declare variables

- 5.4 - Errors when defining variables
- 5.5 - Multiple assignment
- 5.6 - Swapping values
- 5.7 - More about printing
 - 5.7.1 - Printing more than one value
 - 5.7.2 - The write procedures
 - 5.7.3 - Printing Unicode
 - 5.7.4 - Printing to standard error
- 5.8 - General naming conventions

5B - Identifier backslashes

5C – ASCII table

6 – Bool and number types

- 6.1 - Boolean values
 - 6.1.1 Equal values and boolean expressions
 - 6.1.2 Boolean operators
 - 6.1.3 The assert statement
- 6.2 - Number types
 - 6.2.1 - Comparison operators
 - 6.2.2 - Arithmetic operators
 - 6.2.3 - Mixing of different types
 - 6.2.4 - Casting of values
 - 6.2.5 - Autocasting with xx
 - 6.2.5.1 - Cast of bool to int
 - 6.2.5.2 - Cast of int to bool - truthiness
 - 6.2.6 Complex expressions and precedence
 - 6.2.7 Bitwise operators
 - 6.2.7.1 Using bitwise operators
 - 6.2.7.2 Tests on numbers
 - 6.2.8 Formatting procs
 - 6.2.9 Random numbers
 - 6.2.10 The Math module

6B – Times and dates

- 6B.1 – Getting the current time
- 6B.2 - Measuring performance using `get_time` and `current_time_monotonic`
- 6B.3 – Getting a random number from time

7 – Scope of Variables

- 7.1 - Data scope and imperative scope
 - 7.1.1 - Global constants and variables

7.1.2 - Local variables
7.2 - Shadowing of variables

8 – Structuring a project's code

8.1 Structuring with modules
8.2 Loading files with #load
8.3 Named imports
 8.3.1 Definition
 8.3.2 Handling naming conflicts
8.4 Import a file, a dir or a string
8.5 Structuring a project
 8.5.1 The folder structure
 8.5.2 The source code structure

8.6 -import_dir
8.7 Module and program parameters
 8.7.1 Definition and use
 8.7.2 Creating your own module parameters

8B – The #scope directives

8B.1 The #scope_file and #scope_export directives
8B.2 Scope in a module
8B.3 An example of using the #scope directives

9 – More about types

9.1 First class Types
9.2 Constants of type Type: Type alias
9.3 Variables of type Type
9.4 size_of
9.5 The Any type
9.6 Any and the print procedure
9.7 Type comparisons

10 – Working with pointers

10.1 What is a pointer?
10.2 Pointers to pointers
10.3 Dereferencing a null pointer
10.4 Dangling pointers
10.5 Casting to pointer types

10.6 Relative pointers *~snn

11 – Allocating and freeing memory

11.1 The defer keyword

11.2 Allocating and freeing primitive variables

12 - Basics of structs

12.1 Struct declarations

12.2 Making struct variables

12.3 Nested structs

12.4 Struct literals

12.5 Making structs on the heap

12.6 Recursive structs

12.6.1 Linked List

12.6.2 Double Linked List

12.6.3 Tree

12.6.4 Circular dependencies

12.7 A structs namespace

12.8 The #as directive

12.9 Using a structs namespace for better storage management

12.10 Pointer to struct

12.11 Struct alignment

12.12 Making definitions in an inner module visible with using

12.13 Struct parameters

12.13.1 Struct parameters of type Type

12.14 Structs with relative pointers

12.15 Anonymous structs

12.16 Member procs

13 - Unions and enums

13.1 Working with unions

13.2 Working with enums

13.3 Enum as a namespace

13.4 Enum as #specified

13.5 enum_flags and masking flags

13.6 Some useful enum methods

14 - Branching with if else

14.1 The if-else statement

14.1.1 One-liners

- 14.1.2 The classical C error
- 14.2 Ternary operator ifx
- 14.3 Case branching
 - 14.3.1 What is the if-case construct?
 - 14.3.2 Using if-case with enums and #complete
- 14.4 Test on empty variables
- 14.5 Other useful if tests

15 - Looping with while and for

- 15.1 While loop
 - 15.1.1 Nested while loops
 - 15.1.2 Named while-loops
 - 15.1.3 Printing out a recursive list
- 15.2 For loop
- 15.3 Breaking out or continuing a loop
- 15.4 Looping over an enum's values
- 15.5 Runtime-reflection - Looping over a structs fields with type_info()
- 15.6 Serialization
- 15.7 Annotations or notes

16 – Types in depth

- 16.1 Definition of Any, .type and .type.type
- 16.2 Type_Info and Type_Info_Tag
- 16.3 The type_info() proc
- 16.4 Other useful ways to dig into type information
 - 16.4.1 Checking whether an enum is #specified
 - 16.4.2 Checking whether a struct a struct is a subclass of another struct
 - 16.4.3 Type info available at runtime

17 - Basics of procedures

- 17.1 Declaring and calling a proc
 - 17.1.1 Exiting from a proc with return
 - 17.1.2 Getting the type and address of a proc
- 17.2 Local procs
- 17.3 Difference between passing a copy and passing a pointer
- 17.4 Default values for arguments
- 17.5 Named arguments
- 17.6 Multiple return values and #must
 - 17.6.1 Named and default return values
 - 17.6.2 The #must directive
 - 17.6.3 Example proc: file_open

- 17.7 Overloading procedures
 - 17.7.1 What are overloading procedures?
 - 17.7.1 Overloading in global and local scope
- 17.8 Inlining procs
- 17.9 Recursive procs
 - 17.9.1 The #this directive
 - 17.9.2 Recursive structs and #this
- 17.10 Swapping values
- 17.11 A println procedure
- 17.12 Autocasting a parameter with xx
- 17.13 Structs and procs
 - 17.13.1 Using the namespace of a struct in procedures
 - 17.13.2 The #as directive in proc arguments
- 17.14 Reflection on procedures
 - 17.14.1 Getting the argument and return types
 - 17.14.2 The #procedure_name directive
- 17.15 The #deprecated directive
- 17.16 Anonymous procs

18 – Arrays

- 18.1. Array literals
- 18.2. For loop over arrays
- 18.3. Static arrays
 - 18.3.1 Setting up an array with a for loop
 - 18.3.2 Compile-time and run-time bounds check
 - 18.3.3 Using an array as a boolean
 - 18.3.4 Allocating an array on the heap
- 18.4. Dynamic arrays
 - 18.4.1 Useful procs for dynamic arrays
 - 18.4.2 Internal definition of a dynamic array
- 18.5. Array views
 - 18.5.1 Changing the view and the base array
 - 18.5.2 Misuse of array views with dynamic arrays
- 18.6. For-loops over arrays: more examples
 - 18.6.1 Named index and value
 - 18.6.2 Changing an array by iterating with a pointer
 - 18.6.3 Reversing a for loop with <
- 18.7. Multidimensional arrays
- 18.8. Passing an array to a procedure
- 18.9. An array of pointers
- 18.10 Variable number of arguments (..) for a procedure
 - 18.10.1 Passing an array as a variable argument

18.10.2 Named variable arguments proc

18.11 The print procedure

18.12 Array of structs

18B – Ordered remove in arrays

18C - Copying a struct with memcpy

19 – Working with strings

19.1 What are strings?

19.2 Some basic operations on bytes

19.3 Backslash codes, escape characters and Unicode characters

19.4 Some string characteristics

19.4.1 String literals are immutable and bounds-checked

19.4.2 Strings as boolean values

19.4.3 Multi-line strings

19.4.4 Looping over the characters in a string str with for

19.4.5 The sprint procedure

19.4.6 Releasing a string's memory

19.4.7 Storing code in strings

19.4.8 Strings as array views

19.4.9 Relative strings

19.5 String builder

19.6 String operations

19.6.1 Conversions to and from numbers

19.6.1.1 string to numbers

19.6.2 String comparisons

19.6.3 Joining and splitting

19.6.3.1 Looping over the result of a split

19.6.4 Searching

19.6.5 Changing

19.7 C strings

19B - Get command-line arguments

19C - Getting console input

19D - Comparing field names of structs

20 – Debugging

20.1 Some general strategies

20.1.1 Print debugging

20.1.2 Assert debugging

20.2 Debugging compile-time execution

- 20.2.1 #assert debugging
- 20.2.2 The compile-time interactive Jai debugger
- 20.2.3 The #dump directive
- 20.3 Debugging a run-time crash with an external debugger from Visual Studio
- 20.4 Debugging general code
- 20.5 Debugging with natvis
- 20.6 The WinDbg debugging tool
- 20.7 Some general info

21 – Memory Allocators and Temporary Storage

- 21.1 General remarks
 - 21.1.1 Overview of allocation and freeing methods
 - 21.1.2 User defer when possible
 - 21.1.3 Different sorts of memory allocation
- 21.2 Allocators
- 21.3 Temporary storage
- 21.4 Examples of using Temporary Storage
 - 21.4.1 Storing strings in temp with tprint
 - 21.4.2 Storing arrays in temp
 - 21.4.3 Using New with temp
 - 21.4.4 Using Temporary Storage on the Stack
 - 21.4.5 How much memory is allocated in temp?
- 21.5 Memory-leak detector
- 21.6 Check which allocator owns an allocation

22 Polymorphic Procedures

- 22.1 What is polymorphism?
 - 22.1.1 A first example
 - 22.1.2 What is \$T?
- 22.2 Some other examples
 - 22.2.1 T used more than once, and also used as a return type
 - 22.2.2 T as the type of an arrays items
 - 22.2.3 Example with pointers: swapping
 - 22.2.4 Example with structs
 - 22.2.5 Example with several polymorphic types
- 22.3 The lambda notation =>
- 22.4 A procedure as argument of another proc
- 22.5 A recursive lambda as argument of a polymorphic proc
- 22.6 #bake_arguments, \$ and \$\$
- 22.7 A map function

23 Polymorphic arrays and structs

- 23.1 Polymorphic arrays
- 23.2 A more general map procedure
- 23.3 Polymorphic structs
- 23.4 Restricting the type of polymorphic proc arguments
- 23.5 The \$T/Object syntax
- 23.6 The \$T/interface Object syntax
- 23.7 The #bake_constants directive
- 23.8 Polymorphic struct using #this and #bake_constants
- 23.9 Implementing a simple interface
- 23.10 The broadcaster design pattern

23B. Document types: a showcase of inheritance using structs and #as

24 Operator overloading

- 24.1 Operators and operator overloading
- 24.2 Vector operators
- 24.3 Object operators
- 24.4 The #poke_name directive

25 Context

- 25.1 What is the context?
- 25.2 push_context
- 25.3 push_allocator
- 25.4 What does #no_context mean?
- 25.5 Logging
- 25.6 Temporary storage
- 25.7 The stack trace
- 25.8 The print style
- 25.9 Check if a variable is on the stack

26 Meta-programming

- 26.1 The type table
- 26.2 Running code at compile time with #run
 - 26.2.1 The #compile_time directive
 - 26.2.2 The #no_reset directive
 - 26.2.3 Using #run: Computing a struct at compile-time and retrieving at run-time

26.3 Compiling conditionally with `#if`

26.4 Inserting code with `#insert`

26.4.1 Inserting code strings with `#insert`

26.4.2 Inserting compile-time generated strings with `#insert`

26.4.3 Using `#insert` with multi-line strings

26.4.4 Using `#insert -> string`

26.4.5 Type Code and `#code`

26B Macros and `#modify`

26.5 Basics of macros

26.5.1 Using a macro with `#insert`

26.5.2 Using a macro with `#insert` to unroll a for loop

26.5.3 Using a macro for an inner proc

26.5.4 Using a macro with `#insert,scope()`

26.5.5 Using a macro for swapping values

26.5.6 Measuring performance with a macro

26.6 Using a for-expansion macro to define a for loop

26.7 A for-expansion macro for a double linked-list

26.8 A for-expansion macro for an array

26.9 The `#modify` directive

26C Applications of Metaprogramming

26.10 SOA (Struct of Arrays)

26.10.1 Data-oriented design

26.10.2 Making a SOA struct using `#insert`

26.11 How to get the generated source files after the meta-program step?

26.12 How to get info on the nodes tree of a piece of code?

26.12.1 The `compiler_get_nodes` procedure

26.12.2 Changing the AST of code at compile-time

26.13 The `#type` directive and type variants

26.14 Getting the name of a variable at compile time

26.15 Converting code to string

26.16 Creating code for each member in a structure

26.17 A type-tagged union

26.18 Creating code for a list of types

27 Working with Files

27.1 Basic file operations

- 27.2 Working with CSV files
- 27.3 How to copy some files from one folder to another
- 27.4 Deleting subfolders

28 Inline assembly

- 28.1 Inline assembly: what and why
- 28.2 How do Jai and inline assembly interact? - Declaring variables
- 28.3 Some background info
 - 28.3.1 Overview of inline assembly instructions
 - 28.3.2 List of size abbreviations
 - 28.3.3 Assembly Language Data Types
 - 28.3.4 Assembly Feature Flags
 - 28.3.5 List of registers
 - 28.3.6 The Machine_X64 module
- 28.4 Immediate operands
- 28.5 Allocation and pinning
- 28.6 Feature flags
 - 28.6.1 Global build level
 - 28.6.2 Asm-block level
 - 28.6.3 Checking on feature flags
- 28.7 Using AVX and AVX2 SIMD operations
 - 28.7.1 Assembly memory operands: Loading memory into registers
 - 28.7.2 Working with SIMD
- 28.8 Macros and asm
- 28.9 Compile-time execution
- 28.10 Other useful examples
 - 28.10.1 Manipulating an array through pointers
 - 28.10.2 Load Effective Address (LEA) and Load and Read Instruction Example
 - 28.10.3 Fetch and add macro to increment a variable
 - 28.10.4 Binary swap
 - 28.10.5 Reset Lowest Set Bit (BLSR)
 - 28.10.6 Reversing 64-bits integer
 - 28.10.7 Broadcasting, rounding and masking with EVEX

29 Interacting with C

- 29.1 Why would you call C?
- 29.2 How to call C? The #foreign directives
- 29.3 Mapping a dynamic library
- 29.4 Converting a C header (.h) file
- 29.5 Examples on Linux
- 29.6 Examples on Windows
 - 29.6.1 Calling system library functions
 - 29.6.2 Calling user-defined library functions

- 29.7 Callbacks and the `#c_call` directive
- 29.8 Getting the computer name: using `#if`, OS and C interaction
- 29.9 The `Bindings_Generator` module

30 Integrated build system

Intro: What is a metaprogram?

30.1 Workspaces

30.2 The source file location directives

30.3 A minimal build file

- 30.3.1 Compiling with `add_build_file`

- 30.3.2 Compiling with `add_build_string`

- 30.3.3 The `#placeholder` directive

30.4 The build options

- 30.4.1 The optimization level

- 30.4.2 The output type

- 30.4.3 The output executable name

- 30.4.3B The output path

- 30.4.3C The import path

- 30.4.4 The backend options

- 30.4.5 Info about runtime errors and crashes

- 30.4.6 Checks at runtime

- 30.4.7 `runtime_storageless_type_info`

- 30.4.7B Dead code elimination

- 30.4.8 Optimizing LLVM or X64 build

- 30.4.8B Setting machine-level asm options

- 30.4.9 Debug- and Release builds

- 30.4.10 Preventing the output of compiler messages

30.5 Changing the default metaprogram

30B Manipulating the build process

30.6 Intercepting the compiler message loop

30.7 Building and running on successful compilation

30.8 Getting the file to compile from the command-line and inlining

30.9 Building and running with compiler command-line arguments

30.10 Choosing a debug / release build with compiler command-line arguments

30.11 Enforcing coding standards

30.12 Generating LLVM bitcode

30.13 Using notes to do special metaprogramming

30.14 Writing and loading dynamic libraries and `#program_export`

30.15 Adding binary data to the executable

30.16 Simple example using the `Bindings_Generator` module

31 Working with Threads

31.1 Basics of threads

31.2 Thread groups

31.2.1 Concept and basic example

31.2.2 Getting results from the thread group

31.2.3 Determining the number of threads to use

31.2.4 Periodically checking which portion of the work is completed

31.3 Mutexes

31.4 Building a program using OpenGL, macros and threads

31.5 Minimal implementation of Go-style channels

32 Working with processes

32.1 Running a process within a program

32.2 Creating a process

32.3 Writing to a process

32.4 Reading from a process

33 Graphical(GUI) modules

33.1 The GLFW module

33.2 The SDL module

33.3 The GL module

33.4 Direct3D

33.5 The Simp module

33.5.1 A simple window

33.5.2 A bouncing square

33.6 The Getrect module

33.7 The Window_Creation module

34 Other useful modules

34.1 The Sort module

34.2 The Hash_Table module

34.3 The Pool module

34.3.1 Using a Pool

34.3.2 Allocating a struct on a Pool

34.3.3 Using a pool with a macro

34.3.4 Using a flat pool

34.4 The Mail module

35 External modules

35.1 Raylib

36 Plugins

36.1 What are plugins and how to use them?

36.2 How to make your own plugin?

36.3 Plugins in the standard distribution

A description/discussion of some larger programs, in progressive difficulty:

50 The guessing game

50.1 Linux version

50.2 Windows version

51 The Game of Life

51.1 A console print version

51.2 A graphical version

Larger example programs:

23B. Document types: a showcase of inheritance using structs, as and polymorphism

27.2 Deleting subfolders

31.2 Building a program using OpenGL, macros and threads

50. The guessing game

51. The Game of Life