

Appendix C – Directives:

Directives are part of the language, they are not part of a pre-processor as in C.

In the see § references below, the bold reference defines the directive, the other references mostly contains examples of use.

(Count: Jan 2023 - 60)

#add_context	adds a declaration to Context. The main program, and any modules that you use, can add things to the Context: <i>see § 25.1, 25.9</i>
#align	used to align struct member fields relative to the start of the struct: <i>see § 12.11, 18.3</i>
#as	indicates that a struct can implicitly cast to one of its members. It is similar to <code>using</code> , except <code>#as</code> does not also import the names. <code>#as</code> works on non-struct-typed members. For example, you can make a struct with a float member, mark that <code>#as</code> , and pass that struct implicitly to any procedure taking a float argument: <i>see § 12.8, 16.2, 16.3.6, 17.13.2, 23.9, 23B</i>
#asm	specifies that the next statements in a block are inline assembly: <i>see § 28</i>
#assert	does a compile-time assert. This is useful for debugging compile-time meta-programming bugs: <i>see § 20.2.1, 20.1.2, 26.3, 51.2</i>
#bake_constants	generate a compiled procedure with predefined values for type variables: <i>see § 23.7, 23.8</i>
#bake_arguments	provide specific values to a procedure at compile time; does a compile-time currying of a procedure/parameterized struct: <i>see § 22.6, 23.3, 23.7</i>
#c_call	follows C ABI conventions: makes the function to use the C calling convention. Used for interacting with libraries written in C: <i>see § 4.11, 25.1, 23.13, 29.7, 30.14</i>
#caller_location	it gives the line number from where a procedure is called: <i>see § 25.5, 30.2</i>
#caller_code	fills out its Code's parent scope. <code>#caller_code</code> to help create macros that make it easier to implement the kind of thing you might do from an external metaprogram. See <code>how_to/497_caller_code.jai</code> for details: <i>see § 26.14</i>
#char	makes the next one character string after it into a single ASCII character (e.g. <code>#char "A"</code>): <i>see § 5.1.2, 5.3.1, 6.2, 19.1, 19.2</i>
#code	specifies that the next statement/block has type Code : <i>see § 26.4.1, 26.5.6, 26.12, 34.3.3,</i>
#complete	Ensures an if-case statement checks all values of the enum: <i>see § 13.2, 14.3, 14.3.1</i>
#compiler	is a function that interfaces with the compiler as a library; the <code>proc</code> is internal to the compiler: <i>see § 26.12, intro § 30</i>
#compile_time	evaluates to true if execution is occurring during compile time / <code>false</code> during runtime: <i>see § 26.2.1</i>
#cpp_method	allows one to specify a C++ calling convention: <i>see § /</i>
#cpp_return_type_is_non_pod	allows one to specify that the return type of a function is a C++ class, for calling convention purposes (pod = plain old data): <i>see § /</i>
#deprecated	marks a function as deprecated. Calling a deprecated function leads to a compiler warning: <i>see § 17.15</i>
#dump	dumps out the bytecode and basic blocks used to construct the function. This is useful for viewing the disassembly of the bytecode: <i>see § 20.2.3</i>

#expand	marks the proc as a macro: <i>see § 25.9, 26.4.1, 26.5 – 26.8, 28.2, 30.13, 31.4, 33.6, 34.3.3, 51</i>
#file	evaluate to the name of the current source file / path+filename of running executable: <i>see § 30.2</i>
#filepath	path of the currently running executable: <i>see § 30.2, 30.2, 30.6, 30.12</i>
#foreign	instruct compiler to link against a foreign library / specifies a foreign procedure: <i>see § 19.8, 29.2, 29.3, 29.6.2, 29.8, 30.14, 50.2,</i>
#if	Compiling conditionally with #if, #else does not exist, use else: <i>see § 4E, 20.2.1, 22.6, 26.3, 29.8, 31.2.3,</i>
#import	brings a library file into scope / takes foreign modules located in the Jai modules directory and compile the library into your program: <i>see § 8.1, 8.3, 8.4</i>
#import, file	<i>see § 8.4</i>
#import, dir	<i>see § 8.4</i>
#import, string	<i>see § 8.4</i>
#insert	inserts a piece of compile-time generated code into a function or a struct: <i>see § 26.4, 26.5.1, 26.10.2 34.3.3</i>
#insert, scope()	similar to #insert, except it also allows code to access variables in the local scope: <i>see § 26.5.4</i>
#intrinsic	marks a function that is handled specifically by the compiler, like memcpy, memcmp, memset (see Preload.jai): <i>see § 4C</i>
#library	provide a library for the compiler to link against for procedures marked with #foreign directive / specifies file for foreign functions: <i>see § 29.3, 29.6.2, 30.14</i>
#line	evaluate to the line number of the current statement: <i>see § 30.2</i>
#load	bring a source code file into scope / Load source code, as if it were placed right here / takes Jai code files written by the programmer and adds the files to your project: <i>see § 8.1, 8.2, 26.3, 30.3.1</i>
#location	gives the location of a piece of Code: <i>see § 30.2</i>
#modify	used in polymorphic procedures to filter polymorphic parameter type: <i>see § 26.9</i> lets one put a block of code that is executed at compile-time each time a call to that procedure is resolved. One can inspect parameter types at compile-time; Goal: to filter or check on polymorphic parameter type / provide a function to manipulate a type variable prior to it being used in a polymorphic procedure
#module_parameters	specifies the variable as a module parameter: <i>see § 8.7</i>
#must	requires the caller to assign / use the particular return values of the called function. Used primarily for malloc or opening file handles: <i>see § 17.6.2, 19.6.2, 22.2.3</i>
#no_abc	Turn off bounds checking for the scope of a particular array/string access: in this function, do not do array bounds checking: <i>see § 18.3.2, 22.1</i>
#no_alias	<i>see §</i>
#no_context	tells the compiler that the function does not use the context: <i>see § 25.4</i>
#no_debug	tag a macro / macro-call with #no_debug to prevent the compiler from generating any debug line info for that macro: <i>see §</i>

#no_padding	tells the compiler to do no padding for this struct: <i>see § 12.11</i>
#no_reset	lets one store data in the executable's global data, without having to write it out as text: <i>see § 26.2.2</i>
#place	a way of forming a union data type with a struct; Set location in struct of following members: <i>see § 13.1</i>
#placeholder	specifies to the compiler that a particular symbol will be defined/generated by the compile-time metaprogram: <i>see § 30.3.3</i>
#program_export	see Program entry point / used in modules Runtime_Support.jai and Program_Print: <i>see § 4E, 30.14</i>
#procedure_name	gives you the statically-known-at-compile-time name of a procedure: <i>see § 17.14.1</i>
#procedure_of_call	shows what procedure would be called in a particular case of a polymorphic proc <i>see § 22.2.1</i>
#run	execute <code> at compile time (not run time) / takes the function in question and runs that function at compile time: <i>see § 2B, 3.3.4, 4.2, 4B, 5.2.2, 8.7.2, 10.3, 16.3.5, 20.2.2, 26.2.3, 26.4, 30.1, 30.3.3, 30.4.9</i>
#runtime_support	Proc comes from module Runtime_Support: <i>see § 4E, 25.1</i>
#scope_export	makes the function accessible to the entire program: <i>see § 8B.1</i>
#scope_file	makes the function only callable within the current file: <i>see § 8B.1</i>
#scope_module	makes the function only callable within the current module: <i>see § 8B.2</i>
#specified	requires values of an enum to explicitly be initialized to a specific value. An enum marked #specified will not auto-increment, and every value of the enum must be declared explicitly.; Declare intention of maintaining enum values compatibility over time: <i>see § 13.4, 16.3</i>
#string	<i>see § 12.1</i> / #string<token> Parse the next lines as a string up to the next occurrence of the token: used to specify a multi-line string: <i>see § 19.1, 26.4, 27.1, 30.3.2, 34.4</i>
#symmetric	operator overloading (commutativity): allows to swap the 1st and 2nd parameters in a two parameter function. Useful in the case of operator overloading: <i>see § 24.2</i>
#system_library	specifies system file for foreign functions: <i>see § 2.2.6, 19.8, 29.2, 29.8, 50.2</i>
#this	it returns the procedure, struct type, or data scope that contains it as a compile-time constant: <i>see § 17.9, 17.9.1, 17.9.2, 22.5, 23.7, 23.9,</i>
#through	see if-case: case fall-through: <i>see § 14.3</i>
#type	tells the compiler that the next following syntax is a type. Useful for resolving ambiguous type grammar: <i>see § 21.2, 26.13, 29.7, 31.2.1, 51</i>

TYPE VARIANTS:

#type_distinct	<i>see § 26.13</i>
#type_isa	<i>see § 26.13</i>
#type_info_none	marks a struct such that the struct will not generate the type information; Struct does not keep runtime type info: <i>see § 16.3.5</i>

#type_info_procedures_are_void_pointers

makes all the member procedures of a struct void pointers when generating type information. See `Type_Info_Struct_Member.Flags.PROCEDURE_WITH_VOID_POINTER_TYPE_INFO`: *see § 16.3.5*

#type_info_no_size_compliant

prevents the compiler from complaining about the size of the type information generated by a struct: *see §*