Wizardry Language Specification Manual

Standard version 2022041600 (in-development)

Table of Contents

General Syntax

Space, tab, and newline are considered whitespace.

Blocks begin with an opening curly brace.
Blocks end with a closing curly brace and do not require a semicolon.
Labels end with a colon and do not require a semicolon.
All other statements require a semicolon.

Valid names (variable, function, and label) must contain at least one character that is not 0-9, and can include the characters A-Z, a-z, 0-9, and underscore.
Valid names cannot begin with 0x, 0b, or 0B.
Names are case sensitive.

Values are decimal by default. To use a hexadecimal value, add 0x with no whitespace before the hexadecimal digits. To use an octal value, add 0 with no whitespace before the octal digits. To use a binary value, add 0b or 0B with no whitespace before the octal digits.

Strings are treated as one if placed next to each other without an operator.

To call a function, place an opening and closing parenthesis after a variable or value and place the arguments between the parentheses.
Numerical values in front of a function call are assumed to be the type `func:void()` without being cast.

Namespaces are defined in shards meaning that after defining a namespace, subsequent definitions of that namespace are allowed and added on to the first.
Namespace shards are not evaluated after being put together, but are evaluated individually. Outside the namespace, the namespace name and operator are appended to any non-private functions and variables.
Namespace shards do not have to use the namespace name and operator to address functions and variables in the current shard or other namespace shards of the same name. However, functions and variables outside the namespace are given precedence. To ensure the use of the functions and variables defined and/or declared in the current namespace, prefix the namespace name and operator to the function or variable name.

Variables

Variables are declared like so:

```
[type] [name]
```

Variables can be defined by adding an equal sign and the value like so:

```
int myint = 123;
```

Assigning to a previously declared or defined variable is the same but without the type.

To define a structure type, use the `struct` or `structure` statement like so:

```
struct {
     int num1,
     int num2,
     int num3
} mystructtype;
```

Structure variables can be defined the same way as regular variables except the `struct` or `structure` keyword is used to enclose the structure definition name and the values are placed in a block and separated by commas like so:

```
struct(mystructtype) mystruct = {123, 456, 789}
```

The number of values given cannot exceed the number of elements in the structure.
Assigning to a structure is the same but without the type.
Structures can also be assigned to like so:

```
mystruct = {.num1 = 246, .num3 = 357}
```

Or:

```
mystruct.num2 = 159;
```

Arrays can be defined using square brackets and assigned to the same ways as structures:

```
int myarray[3] = {123, 456, 789}
```

Place the size of the array between the brackets. A size of zero is invalid.

To get an element from a structure pointer, place a dollar sign after the variable name:

```
mystruct$.num1 = 124;
```

To declare or define an array, place square brackets and the number of elements after the type name like so:

```
int[3] myarray;
int[3] myarray2 = {1, 2, 3};
```

Values are placed in a block and separated by commas. The number of values should not exceed the number of elements in the array.

Multiple array dimensions can be declared or defined like so:

```
int[2][3] myarray;
int[2][3] myarray2 = {{1, 2, 3}, {4, 5, 6}};
```

This will multiply the array size.

Pointers can be declared or defined by placing a dollar sign after the type name like so:

```
int$ mypointer;
int$ mypointer2 = 123;
```

To get a pointer to a variable, put a dollar sign in front of the variable name like so:

```
$myvar
```

To get data at a pointer, put a dollar sign after the variable name like so:

```
myvar$
```

Multiple dollar signs can be added to help in the case of a multi-level pointer.

Pointer and array suffixes can be used together.
Make an array of pointers to integers like so:

```
int[5]$ myvar;
```

Make a pointer to an array of integers like so:

```
int$[5] myvar;
```

Functions

Functions are declared like so:

```
[type] [name] ([arg type] [arg name])
```

Arguments are separated by commas

To declare a function to be defined later, add a semicolon like so:

```
int main(int, char$$);
```

As shown above, the variable name is optional.
Function declaration is optional as a function definition also counts as a declaration.
You can declare a function multiple times.

To define the function, add an opening curly brace, place your code, and end if with a closing curly brace like so:

```
int main(int argc, char$$ argv) {

}
```

The variable name is mandatory when defining a function.
You can only define a function once.

Declare and define a variadic function like so:

```
void putFString(char$, ...);
void putFString(char$ format, ...) {

}
```

The three periods indicate that the function is variadic and accepts any number of arguments.
No more arguments should be defined after the three periods.

Operators

Arithmetic	
+	Add.
-	Subtract.
*	Multiply.
/	Divide.
%	Modulus.

Comparison	
==	Equal to.
>	Greater than.
<	Less than.
>= =>	Greater than or equal to.
<= =<	Less than or equal to.
!=	Not equal.

Logical	
&&	AND.
||	OR.
!	NOT.

Bitwise	
&	AND.
|	OR.
~	NOT.
^	Exclusive OR.
<<	Left shift.
>>	Right shift.
<<<	Left rotate.
>>>	Right rotate.

Assignment	
=	Assign the right to the left.
+=	Add the right to the left.
-=	Subtract the right from the left.
*=	Multiply the left by the right.
/=	Divide the left by the right.
%=	Assign the modulus of the left, using the right, to the left.

Miscellaneous	
$	Returns a pointer when placed before a function, variable, or value name. Returns the data at a pointer when placed after a variable that is a pointer.
++	Increment before return if placed before variable name and increment after return if placed after variable name. Extra plus symbols may be added to increase the number of increments.
--	Decrement before return if placed before variable name and decrement after return if placed after variable name. Extra minus symbols may be added to increase the number of decrements.
.	Access namespace and structures. Place after a namespace or structure name to access functions and variables of that namespace or elements of that structure.

Types

Modifier prefixes and suffixes must be added without whitespace while behavior prefixes must be added with whitespace.

Base types	
`void`	Invalid if used to declare a variable and means no return value if used to declare a function.
`int`	32-bit signed integer.
`char`	8-bit signed integer.
`float`	32-bit floating point number.
`bool`	int-sized boolean by default but size can be changed to char-sized. This should not be used without a prefix in modular code such as libraries due to the possibility of a size difference.
`i8`	Signed 8-bit integer.
`i16`	Signed 16-bit integer.
`i32`	Signed 32-bit integer.
`i64`	Signed 64-bit integer.
`u8`	Unsigned 8-bit integer.
`u16`	Unsigned 16-bit integer.
`u32`	Unsigned 32-bit integer.
`u64`	Unsigned 64-bit integer.
`f32`	32-bit floating point number.
`f64`	64-bit floating point number.

Optional types	
`i24`	Signed 24-bit integer.
`i48`	Signed 48-bit integer.
`i128`	Signed 128-bit integer.
`u24`	Unsigned 24-bit integer.
`u48`	Unsigned 48-bit integer.
`u128`	Unsigned 128-bit integer.
`dec32`	32-bit decimal floating point number.
`dec64`	64-bit decimal floating point number.
`f16`	16-bit floating point number. If this is available then the `s` prefix should make `float` 16-bit.

Modifier prefixes	
`u`	Makes `int` and `char` unsigned and is invalid on other types.
`l`	Does not effect `int` on 32-bit architectures, makes `int` 64-bit on 64-bit architectures, makes `float` 64-bit, forces `bool` to be int-sized and is invalid on other types.
`ll`	Makes `int` 64-bit, makes `float` 64, 80, or 128-bit dependent on architecture, and is invalid on other types.
`s`	Makes `int` 16-bit, forces `bool` to be char-sized, and is invalid on other types.

Modifier suffixes	
`$`	Makes any type a pointer to data of that type and can be used on `void` to make it a pointer to any type. Repeating will increase the pointing depth ($$ is a pointer to a pointer, $$$ is a pointer to a pointer to a pointer, etc.)
`[]`	Makes any type a pointer to an array and can be used on `void` if not used as the last modifier suffix. A number can be specified inside to specify the array size. The number will be cast to an integer of machine address bus size.

Special types

`func`	Defines a function pointer with a certain return value and arguments. Variadic function pointers can be defined by three periods after the last non-optional argument type. Syntax: `func:[return type]([comma separated argument types])` Example: `func:int(int, char$)`

`asm`	Invalid if used to declare a variable and makes it so a function's code is to be written in assembly. Syntax names are defined by the compiler. The default syntax is also compiler-specific. For assembly code that interferes with Wizardry's characters, place the code inside quotes instead of curly braces and end with a semicolon.
	
	Syntax: `asm:[return type]:[optional asm syntax]`
	
	Example:
	`asm:void test() {`
	`  mov eax, 1`
	`  mov ebx, 2`
	`}`
	`asm:void test2() "\`
	`  mov eax, 1\n\`
	`  mov ebx, 2\`
	`";`

Keywords

Program Flow		
`break`	`break;`	Exits the current statement, `do`, or `while` block or exits a case statement.

`namespace`	`namespace <name> {`
	`  <statements>`
	`}`
		Defines a namespace shard appending the namespace name and operator to all functions or variables defined inside.

`continue`	`continue;`	Skips the remaining code in a `do` or `while` block.
`do`	`do (<condition>) <statement or statement block>`	Runs code then loops as long as the condition is true.
`else`	`else <statement or statement block>`	Can only be put after an `if` or `elseif` block and runs code if all previous conditions in the chain are false.
`elseif`	`elseif (<condition>) <statement or statement block>`	Can only be put after an `if` or another `elseif` block and runs code if all previous conditions in the chain are false and the current condition is true.
`fall`	`fall <case value>;`	Falls through to another case inside of a switch statement and falls through to the case directly under if no case is provided.
`for`	`for (<variable declarations>; <condition>; <each-time statement or block>) <statement or block>`	Declares the variables specified into the statement or block and for each time the condition evaluates to non-zero, runs the statement or block then the each-time statement or block.
`goto`	`goto <label>;`	Jumps to a label inside of the current function.
`if`	`if (<condition>) <statement or statement block>`	Runs code if the condition is true. Removing the brackets makes the block effective for only 1 statement after.
`return`	`return <data>;`	Returns from a function and returns data if not `void`.

switch	switch (<variable>) {
	  case <value>:
	  <code>
	  case <value>..<value>:
	  <code>
	  case <value>...<value>:
	  <code>
	  default:
	  <code>
	}
		Compares a variable and jumps to the code at the corresponding `case` statement. The code at the default statement will run if provided and no `case` statements match. Putting two dots between two values will match that case for the first value and any values in-between. Putting three dots between two values will match that case for the first value, any values in-between, and the last value.

while	while (<condition>) <statement or statement block>	Loops code as long as the condition is true.

Assignment		
cast	cast(<data>, <type>)	Returns data as a different type.
ternary	ternary(<condition>, <value 1>, <value 2>)	Returns value 1 if the condition is true and value 2 otherwise.

Data		

enum	enum (<equation>) {
	  <name>;
	  <name> = <rhs value>;
	}
		Makes an enumerated variable list and the variables will not receive symbols. Placing an equation in parentheses with the format `(lhs operator rhs)` will change what is used to generate the numbers. The default is `(0 + 0)`. Specifying parts of the equation will only change that part. If only one value is specified, it is assumed to be the left hand side. Values are generated by incrementing the right hand side. Placing an equal sign and a value after a name will set the right hand side to that value and continue incrementing from there.

newtype	newtype <name> <existing type or structure definition>	Creates a new type using a predefined type or structure definition
sizeof	sizeof(<type or variable>)	Returns the size in bytes of a type or the type of a variable.

struct structure	struct <name> {
	  <variable declaration>;
	}
	
	struct(<name>) <variable declaration(s) or definition(s)>
		Defines a structure definition or creates a structure variable.

typeof	typeof(<variable>)	Behaves as though you typed in the type name of the given variable.

[type]: [op]	<type>:<operation> (<var>, <types> <val>) <statement or statement block>	Defines code to preform an action on a type. Internal types are invalid. If no statements are provided, then the operation between the main type and the types specified for the value argument are considered invalid. If no types are provided to the value argument, the statements will be applied as if every available type was provided. The return statement expects to return the same type as the main type.
		
		Operations: add, and, cast, dec, div, inc, lrot, lshift, mod, mul, or, rrot, rshift, xor.
		
		Example:
		struct(mystruct):add(input, __INT_TYPES value) {
		  return input.data + value;
		}

Miscellaneous

`asm: [syntax]`	`asm:<optional syntax> (<quoted register name> <variable name to set register to>) { <assembly> } asm:<optional syntax> (<quoted register name> <variable name to set register to>) "<assembly>";`	Compile assembly into the current position. If no colon and syntax name are provided, the compiler should use its default syntax. The parentheses are also optional and can be omitted if no variables are to be written to registers. For assembly code that interferes with Wizardry's characters, place the code inside quotes instead of curly braces and end with a semicolon.
`attrib`	`attrib (<attribute(s)>) <statement or statement block>`	Set attributes for a statement or statement block. Attributes are a comma-separated list of `attrib = value`. Attributes can only be set to static values.
`defer`	`defer <statement or statement block>`	Sets the code to be run before exiting the current scope.
`defer:add`	`defer:add <statement or statement block>`	Adds to the code to be run before exiting the current scope.
`defer:addtop`	`defer:addtop <statement or statement block>`	Adds to the top of the code to be run before exiting the current scope.
`vararg`	`vararg(<type>)`	Returns the next argument in a variadic function. Is invalid in functions that are not variadic.

Modifiers		
`const constant`	`const <variable declaration and/or definition>`	Declares a variable as a constant and makes assignment to that variable illegal. This is invalid on functions.
`extern external`	`extern <variable declaration and/or definition>`	Declares a variable that is stored outside the current file and is invalid on functions.
`inline`	`inline <function declaration and/or definition>`	Makes it so a function can be embedded into where it is called from to avoid a function call and increase speed. This is invalid on variables.
`private`	`private <function declaration and/or definition>`	Makes it so a function or variable is not visible outside of the current file or namespace shard.

`reg` `register`	`reg <variable declaration and/or definition>`	Recommends that the compiler use a register to hold the data and is invalid on functions.
`static`	`static <variable declaration and/or definition>`	Makes a local variable hold its value after exiting its scope and is invalid on functions.
`volatile`	`volatile <variable declaration and/or definition or function declaration and/or definition>`	Makes a variable or function ineligible for optimization.

Preprocessor Commands and Functions

All preprocessor commands should start with an at symbol.
Preprocessor commands end after a newline.
To continue a command past a newline, insert a backslash before the newline.
Preprocessor commands can only accept static values unless stated othwise.

#	# <text>	Start comment until end of line.
#<	#< <text> <text> <text> ># #< <text> >#	Start a multi-line comment.
>#	#< <text> <text> <text> ># #< <text> >#	End a multi-line comment.
else	@else <statements>	Exposes the statements provided if the above elseif or if command is false.
elseif	@elseif <condition> <statements>	Exposes the statements provided if the condition is true and the above conditional command is false.
elseifdef	@elseifdef <macro names> <statements>	Exposes the statements provided if the provided macro names are defined and the above conditional conditional command is false. The ||, &&, and ! operators along with parentheses can be used to test multiple names.
endif	@endif	Ends an if command.
def define	@define <macro name> <text>	Defines a macro as the text provided.
defined	defined(<macro name>)	Returns (1) if a macro is defined and (0) otherwise.
defifndef	@defifndef <macro name> <statements>	Defines a macro as the text provided if it is not already defined.
if	@if <condition> <text>	Exposes the text provided if the condition is true.

ifdef	@ifdef [macro names] [statements]	Exposes the statements provided if the provided macro names are defined. The `||`, `&&`, and `!` operators along with parentheses can be used to test multiple names.
include	@include [file name]	Includes a file onto the current line. <file>: Include file from the standard include directory. If no file extension is provided, search for the file name and if there is no match, search for the file name with .wh added. [file]: Solve the file name as a filename regular expression and include it from the standard include directory. 'file': Include file relative to the current file's location. If no file extension is provided, search for the file name and if there is no match, search for the file name with .wh added. "file": Solve the file name as a filename regular expression and include it relative to the current file's location.
pragma	@pragma [command]	Sends a command to the compiler.

Compiler Pragma Commands

`overload`	`overload [true/yes or false/no]`	Allows function overloading and enables function name mangling for overloaded functions. This is false by default.
`lazybool`	`lazybool [true/yes or false/no]`	Causes `bool` to not guard against becoming greater than 1 or less than 0. This is false by default.
`smallbool`	`smallbool [true/yes or false/no]`	Causes `bool` become the same size as char instead of the same size of int. This is false by default.
`var variable`	`var [comma separated list of [var = val]]`	Sets internal compiler variables.
`zeroinit`	`zeroinit [true/yes or false/no]`	Causes local variables to be set to zero instead of uninitialized when no initial value is given. This is true by default.
`...`	`...`	Compiler-specific commands can be added.

Compiler Macros

All compiler-defined macros except `NULL`, `true`, and `false`, should start with two underscores. The compiler should not define any functions or variables.

`__ARCH_[name]`	Name of the CPU architecture(s) the compiler is targeted at. If the architecture has multiple names, most known names and combinations should be available.
	
	Examples:
	`__ARCH_x86_64`
	`__ARCH_amd64`
	`__ARCH_AMD64`
	
	`__ARCH_x86`
	`__ARCH_i386`
	
	`__ARCH_PSIS32`
	`__ARCH_PSIS_32`

`__ASM_DEFAULT_[name]`	Name of the assembly syntax or assembly compiler the compiler uses by default. Example: `__ASM_DEFAULT_ATT`

`__ASM_HAS_[name]`	Name of the assembly syntax(s) or assembly compiler(s) the compiler supports. For general syntax support, the macro should use the name of the syntax. For support of a specific compiler, the macro should use the compiler's name.
	
	Examples:
	`__ASM_HAS_GAS`
	`__ASM_HAS_ATT`
	`__ASM_HAS_INTEL`
	`__ASM_HAS_NASM`

`__COMPILER [name]`	Name of the compiler in an 8-bit ASCII string. Example: `"Wizard"`
`__COMPILER_BUILD [number]`	Build or version number of the compiler. Example: `10210`

`__COMPILER_BUILD_STRING [string]`	Build or version of the compiler in a string format. Example: `"1.2.10"`
`false (0)`	A value that evaluates as false.
`__FILE`	A dynamic macro containing the name of the current file contained in a string. Example: `"hello.wiz"`
`__FUNCTION`	A dynamic macro containing the name of the current function contained in a string. When used globally, the string should be empty. Example: `"main"`
`__FLOAT_TYPES`	Should be a space-separated list of floating-point type names supported by the compiler. This should always include all variations of the `float` base type and any of the base types starting with `f`. This should not include decimal floating point types such as `dec32` and `dec64`.
`__INT_TYPES`	Should be a space-separated list of integer type names supported by the compiler. This should always include all variations of the `int` and `char` base types and any of the base types starting with `i` or `u`.
`__LINE`	A dynamic macro containing the line in the file that this macro was used. Example: `47`
`__NAMESCACE`	A dynamic macro containing the name of the current namespace contained in a string. When used outside of any namespaces, the string should be empty. Example: `"wiz"`
`__NAMESCACE_TREE`	A dynamic macro containing the current namespace tree (contained in a string. When used outside of any namespaces, the string should be empty. Example: `"mylib.io.funcs"`

`NULL`	A null pointer. This should be `(cast(void$, 0))` for most platforms. For platforms that do not use zero to indicate a null pointer, only the number should be changed.
`__OPTIONALTYPE_[name]`	Defined if the compiler supports an optional type with this name. Example: `__OPTIONALTYPE_dec64`

`__OS_[name]`	Name of the operating system the compiler is targeted at in capital letters. If the operating system has multiple names or terms, most known names and combinations should be available.
	
	Examples:
	`__OS_WINDOWS`
	`__OS_WINDOWS_NT`
	`__OS_WINDOWS_9x`
	
	`__OS_GNU_LINUX`
	`__OS_LINUX`
	`__OS_GNU`
	
	`__OS_PHOSPHOROS`
	`__OS_PHOS`

`__STANDARD [version]`	Version number of the Wizardry standard the compiler supports. Example: `2022041301`
`true (1)`	A value that evaluates as true.

Grammar

Grammar syntax:

- `:` : Make definition
- `<>`: Insert definition
- `()` : Group
- `[]`: Optional group or definition
- `{}`: Choice
- `\\`: Non-literal description
- `"”`: Interpreted string (supports backslash escaped characters)
- `‘’` : Literal string
- `//`: Regular expression (must match from start to end of input to be considered a match)
- `|`: Or
- `…`: Continue group

```
code: [{[space] <statement> [space] | [space] ‘{’ [space] [code]…
[space] ‘}’ [space] | /^[ \t]*/ <preproc cmd> /[ \t]*$/}]

space: (‘ ’ | “\t” | “\n”)…

name: [\String of A-Z, a-z, 0-9, _ not starting with 0x, 0b, or
0B\] \String of A-Z, a-z, _\ [\String of A-Z, a-z, 0-9, _\]

namespace: (<name> ‘::’)…

number: /(?<![.])(0x[0-9a-fA-F]+|0[bB][0-1]+)(?![0-9.])|([-]?(?<![0-
9xbB\.])([0-9]+\.|[0-9]*\.?[0-9]+)(?![0-9xbB.])([eE][-+]?[0-9]+)?)/

string: ‘"’ \String of ASCII chars 1-33, 35-91, 93-255, or a
backslash before a backslash, n, r, b, e, t, v, ", x and 2 hex
digits, or 1-3 decimal digits\ ‘"’

char: ‘'’ \ASCII char 1-38, 40-91, 93-255, or a backslash before a
backslash, n, r, b, e, t, v, ', x and 2 hex digits, or 1-3 decimal
digits\ ‘'’

regular operator: {‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘%’ | ‘=’ | ‘+=’ | ‘-=’ |
‘*=’ | ‘\=’ | ‘%=’ | ‘==’ | ‘>’ | ‘<’ | ‘<=’ | ‘=<’ | ‘>=’ | ‘=>’ |
‘&&’ | ‘||’ | ‘&’ | ‘|’ | ‘~’ | ‘^’ | ‘<<’ | ‘>>’ | ‘<<<’ | ‘>>>’}

inc/dec operator: {“++” [‘+’]… | “--” [‘-’]…}

variable: [‘!’] {[inc/dec operator] [‘$’] [namespace] <name> | [‘$’]
[namespace] <name> [inc/dec operator]} [‘$’]…
```

```
value stub: {<string> | <char> | <number> | <variable> | <func call>
| ‘(’ <value> ‘)’ | ‘{’ <value> [[space] ‘,’ [space] <value>]… ‘}’}

value: <value stub> [<regular operator> <value>]

type: \type name matching <name>\

decl: <type> <space> <name>

var decl stub: <name> [[space] ‘=’ [space] <value>]

var decl: <type> <space> <var decl stub> [[space] ‘,’ [space] <var
decl stub>]…

func decl arg: <type> [<space> <name>]

func decl: <type> <name> [space] ‘(’ [space] [<func decl arg>
[[space]‘,’ [space] <decl>]…] [space] ‘)’

func def: <type> <name> [space] ‘(’ [space] [<decl> [[space]‘,’
[space] <decl>]…] [space] ‘)’ [space] ‘{’ [code]… ‘}’

func call: {[namespace] <name> | <value stub>} [space] ‘(’ [space]
[<value> [[space] ‘,’ [space] <value>]…] [space] ‘)’

statement stub: {[{<func call >}] [space] ‘;’ | [{<func def>}]}

preproc cmd: ‘@’ [space] \command and arguments\
```

Standard Library

All standard library functions, variables, and type definitions should be held in the `wiz` namespace.

Standard library headers should be split into separate headers and a `wizardry.wh` header should be provided to include all the headers under `wizardry/*.wh`.

Any variables, functions or defines added to the standard library should start with an underscore. It is recommended that any types added should start with an underscore.
OS or API specific variables, functions, types and defines should be placed in a separate header file or separate header files in a folder named with the prefix (usually name or initials). The variables, functions, types and defines should start with the prefix of the OS or API and an underscore.

Macros and variables missing a value next to them are defined based on implementation.

wizardry/	
Variables	
Functions	
Types	
Macros	

wizardry/io.wh	
Variables	
`wiz.file wiz.in`	Input stream file pointer.
`wiz.file wiz.out`	Output stream file pointer.
`wiz.file wiz.err`	Error stream file pointer.
`int wiz.infd = 0`	Input stream file descriptor.
`int wiz.outfd = 1`	Output stream file descriptor.
`int wiz.errfd = 2`	Error stream file descriptor.
Functions	

`int wiz.close(int fd)`	Close a file stream using a descriptor. Returns 1 on success and on failure, returns 0 and sets `wiz.error`. `fd`: File descriptor to close.
`int wiz.fclose(int fileptr)`	Close a file structure by closing the stream with `close` and deallocating the structure using `free`. Returns 1 on success and on failure, returns 0 and sets `wiz.error`. `fileptr`: Pointer to file structure to close.
`wiz.file wiz.fopen(char$ path, int flags, wiz.filemode mode)`	Uses `open` to open a file descriptor and returns a pointer to a `file` structure dynamically allocated using `alloc`. Returns a valid pointer on success and on failure, returns `NULL` and sets `wiz.error`. `path`: File path. `flags`: Flags to open the descriptor. `mode`: File mode flags.
`int wiz.fdprint(int fd, char$ text)`	Writes a string to a file descriptor. `fd`: File descriptor. `text`: String to write.
`int wiz.fdprintf(int fd, char$ text, ...)`	Writes a formatted string to a file descriptor. `fd`: File descriptor. `text`: String to write. `...`: Variadic arguments for format.
`int wiz.fprint(wiz.file file, char$ text)`	Writes a string to a file type. `file`: File type `text`: String to write.
`int wiz.fprintf(wiz.file file, char$ text, ...)`	Writes a formatted string to a file type. `file`: File type `text`: String to write. `...`: Variadic arguments for format.

`int wiz.open(char$ path, int flags, wiz.filemode mode)`	Opens a file stream using the lowest possible non-negative descriptor. Opening the same file will return a new descriptor. Returns a descriptor on success and on failure, returns -1 and sets `wiz.error`. `path`: File path. `flags`: Flags to open the descriptor. `mode`: File mode flags.
`int wiz.print(char$ text)`	Writes a string to the output file stream. `text`: String to write.
`int wiz.printf(char$ text, ...)`	Writes a formatted string to the output file stream. `text`: String to write. `...`: Variadic arguments for format.
Types	
`wiz.file`	Implementation-specific.
`wiz.filemode`	Implementation-specific.
Macros	
`WIZ_IN_FILENO 0`	Input stream file number.
`WIZ_OUT_FILENO 1`	Output stream file number.
`WIZ_ERR_FILENO 2`	Error stream file number.
`WIZ_FILE_READ`	File mode flag for reading only.
`WIZ_FILE_RW (WIZ_FILE_READ | WIZ_FILE_WRITE)`	File mode flag for reading and writing.
`WIZ_FILE_WRITE`	File mode flag for writing only.
`WIZ_OPEN_BINARY`	Flag for `open`/`fopen` to open the file without modifying the stream. This may not have any effect depending on the implementation.
`WIZ_OPEN_CREATE`	Flag for `open`/`fopen` to attempt to create the file if it doesn't exist.
`WIZ_OPEN_...`	Implementation-specific flags for `open`/`fopen` can be defined.

wizardry/dynamic.wh
Functions

`int wiz.freesym(wiz.dynlib lib, char$ name)`	Frees a symbol from memory. Any calls to the pointer returned by `_getsym` will result in undefined behavior. Returns 0 on failure and 1 on success. `lib`: Pointer returned by `loadlib`. `name`: Symbol name
`void$ wiz.getsym(wiz.dynlib lib, char$ name)`	Returns the pointer to a symbol from a library or `NULL` if not successful. `lib`: Pointer returned by `loadlib`. `name`: Symbol name
`wiz.dynlib wiz.loadlib(char$ path, int options)`	Opens a dynamic library and returns a pointer to be used with other functions. Searches relative to the current directory by default. Returns `NULL` and sets `wiz.error` if the library cannot be found or an error occurred when attempting to load. `path`: Path or filename of the library file. `options`: Option flags for finding or loading the library. Combine flags with the bitwise OR operator.
`int wiz.unloadlib(wiz.dynlib$ ptr)`	Closes and frees a dynamic library. Returns 1 on success and 0 on failure. `ptr`: Pointer returned by `loadlib`.
Types	
`wiz.dynlib`	Implementation-specific.
Macros	

`WIZ_DYNAMIC_EXTENSION`	File extension of shared libraries native to the target operating system or architecture.
	
	Examples:
	`".so"`
	`".dll"`

`WIZ_DYNAMIC_SEARCH_STANDARD`	Search in the standard library storage paths (`/lib` and/or `/usr/lib` on Unix-based operating systems for example).
`WIZ_DYNAMIC_LOAD_LAZY`	Load in only the parts of the library requested instead of the whole library. This will increase the time it takes to get a symbol but will save memory.

`WIZ_DYNAMIC_...`	Other operating system or architecture specific flags can be added.

`wizardry/memory.wh`	
Functions	
`void$ wiz.alloc(long size)`	Dynamically allocate a block of memory on the heap. Returns a valid pointer on success and on failure, returns `NULL` and sets `wiz.error`. `size`: Size in bytes of how large the block should be.
`void wiz.copymem(void$ src, void$ dest, long len)`	Copies a certain amount of bytes from a memory pointer to another. `src`: Pointer to memory to reading from. `dest`: Pointer to memory to start writing at. `len`: Amount of bytes to copy.
`void wiz.fillmem(void$ ptr, long len, u8 value)`	Writes a certain amount of bytes at a memory pointer. `ptr`: Pointer to memory to start writing at. `len`: Amount of bytes to write. `value`: Value to write.
`int wiz.free(void$ ptr)`	Free a dynamically allocated block of memory returned by `alloc`. If given an invalid pointer, it is recommended to return 0 and set `wiz.error`, but undefined behavior can be exhibited instead. Returns 1 on success. `ptr`: Pointer to the block of memory to be freed.
`void$ wiz.realloc(void$ ptr, long size)`	Reallocates or resizes a dynamically allocated block of memory returned by `alloc`. If given an invalid pointer, it is recommended to return `NULL` and set `wiz.error`, but undefined behavior can be exhibited instead. Returns a valid pointer on success. `ptr`: Pointer to existing memory block. `size`: New size.

`void$ wiz.valloc(long size, u8 value)`	Calls `alloc`, sets the bytes in the memory block returned to a value if `alloc` returns a valid pointer, and returns the output of `alloc`. `size`: Size in bytes of how large the block should be. `value`: Value to set each byte to.
`void$ wiz.zalloc(long size)`	Calls `alloc`, zeros the memory block returned if `alloc` returns a valid pointer, and returns the output of `alloc`. `size`: Size in bytes of how large the block should be.
`void wiz.zeromem(void$ ptr, long len)`	Writes a certain amount of zeros at a memory pointer. `ptr`: Pointer to memory to start writing at. `len`: Amount of zeros to write.
Macros	
`WIZ_FREE_SAFE_INVAL`	Defined if the `free` implementation is protected against invalid pointers.
`WIZ_FREE_SAFE_NULL`	Defined if the `free` implementation is protected against `NULL`.
`WIZ_REALLOC_SAFE_INVAL`	Defined if the `realloc` implementation is protected against invalid pointers.
`WIZ_REALLOC_SAFE_NULL`	Defined if the `realloc` implementation is protected against `NULL`.