Wizardry Language Specification Manual

Standard version 2023051200 (in-development)

Table of Contents

General Syntax

The file extension `.wiz` is suggested for code files.
The file extension `.wh` is suggested for header files.

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.
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.
Without being cast, the type of numerical values in front of a function call are assumed by what the function return value is assigned to and what arguments are provided.

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

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[3] myarray = {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 (equivalent to `&myvar` in C):

```
$myvar
```

To get data at a pointer, put a dollar sign after the variable name like so (equivalent to `*myvar` in C):

```
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 5 pointers to integers like so:

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

Make a pointer to an array of 5 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.

Namespaces

Namespaces are declared like so:

```
namespace <name> {
    [statements]
}
```

Anything you define in a namespace will have the namespace name and a period put before it.

Namespaces can be nested like so:

```
int test_var;
namespace test {
    int a_var;
    namespace my_ns1 {
        int my_var;
        void set_var(int val) {
            test_var = val;
            my_var = val;
            a_var = val;
        }
    }
    namespace my_ns2 {
        int my_var;
        void set_var(int val) {
            global.test_var = val;
            test.my_ns2.my_var = val;
            test.a_var = val;
        }
    }
}
```

To access an element in a namespace, you can either use the name without the namespaces or use the namespace tree. Local elements will take precedence over elements higher up. Name conflicts can be resolved by prepending the namespace tree to the element name or using `global`.

Operators:

Logical operators will cast the conditions being tested to bool.

Assignment and Arithmetic		
`=`	`var = val`	Assign.
`?`	`? (cond) {val1, val2}`	Ternary operator. Resolves to `val1` if `cond` is true and `val2` if `cond` is false.
`+`	`val1 + val2`	Add.
`++`	`++[+...]var` `var++[+...]`	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.
`+=`	`var += val`	Add `val` to `var`.
`-`	`val1 - val2`	Subtract.
`--`	`--[-...]var` `var--[-...]`	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.
`-=`	`var -= val`	Subtract `val` from `var`.
`*`	`val1 * val2`	Multiply.
`*=`	`var *= val`	multiply `val` by `var`.
`/`	`val1 / val2`	Divide.
`/=`	`var /= val`	Divide `val` by `var`.
`%`	`val1 % val2`	Modulus.
`%=`	`var %= val`	Assign the modulus of `var`, using `val`, to `var`.

Comparison		
`==`	`val1 == val2`	Equal to.
`>`	`val1 > val2`	Greater than.
`<`	`val1 < val2`	Less than.
`>=` `=>`	`val1 >= val2` `val1 => val2`	Greater than or equal to.
`<=` `=<`	`val1 <= val2` `val1 =< val2`	Less than or equal to.
`!=` `<>`	`val1 != val2` `val1 <> val2`	Not equal.

Logical		
`&&`	`cond1 && cond2`	Logical AND.
`||`	`cond1 || cond2`	Logical OR.
`^^`	`cond1 ^^ cond2`	Logical XOR.
`!`	`!cond`	Logical NOT.

Bitwise		
`&`	`val1 & val2`	Bitwise AND.
`|`	`val1 | val2`	Bitwise OR.
`^`	`val1 ^ val2`	Bitwise XOR.
`~`	`~val`	Bitwise NOT.
`<<`	`val1 << val2`	Left shift.
`>>`	`val1 >> val2`	Right shift.
`<<<`	`val1 <<< val2`	Left rotate.
`>>>`	`val1 >>> val2`	Right rotate.

Data		
`$`	`$var` `var$`	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.

`.`	`namespace.var`
	`struct.var`
	`{.var = val}`
		Access namespaces, structures, and unions. Place after a namespace, structure, or union name to access functions and variables of that namespace or elements of that structure or union. If used in curly brackets in the format of `.<name> = <val>` being assigned to a struct, only the names specified are assigned (for example, `mystruct = {.data1 = 123, .data3 = 789}` will only assign to `data1` and `data3` of `mystruct`) while the rest of the elements are set to their default value.

Types:

Modifier prefixes and suffixes must be added without whitespace while behavior keywords must be added before the type and with whitespace. Modifier prefixes and suffixes are only valid on built-in types. The compiler should probably treat these as individual type names.

Base types	
`void`	Invalid if used to declare a variable and means no return value if used to declare a function.
`int`	Integer (32-bit signed by default).
`iptr`	Pointer-sized signed integer.
`uptr`	Pointer-sized unsigned integer.
`char`	8-bit integer (signed by default).
`float`	Floating point number (32-bit by default).
`bool`	Boolean (32-bit by default). 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.
`varargs_t`	Pointer to a function’s variadic argument data.

Optional types	
`i24`	Signed 24-bit integer.
`i48`	Signed 48-bit integer.
`i128`	Signed 128-bit integer.
`i256`	Signed 256-bit integer.
`i512`	Signed 512-bit integer.
`u24`	Unsigned 24-bit integer.
`u48`	Unsigned 48-bit integer.
`u128`	Unsigned 128-bit integer.
`u256`	Unsigned 256-bit integer.
`u512`	Unsigned 512-bit integer.
`dec32`	32-bit decimal floating point number.
`dec64`	64-bit decimal floating point number.
`dec128`	128-bit decimal floating point number.
`f16`	16-bit floating point number. If this is available, 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.
`fast`	Overrides the type's size to make it larger if it means an increase in speed. This can be combined with other prefixes and must be put first.

Modifier suffixes	
`!`	Makes the type volatile at the point it is used (`int!$` would be a volatile pointer to an `int`, `int$!` would be a pointer to a volatile `int`, and `int!$!` would be a volatile pointer to a volatile `int`).
`$`	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 the data constant at the point it is used (`int%$` would be a constant pointer to an `int`, `int$%` would be a pointer to a constant `int`, and `int%$%` would be a constant pointer to a constant `int`). Attempting to assign to constant data will generate an error. This should only be used when the data is in a location that is not writable (**.** `rodata` for example).
`[]`	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 unsigned integer of machine address bus size. If a number is not given, it is invalid to use another array suffix directly after without a number (`int[][5][3]` is valid, `int[][5][]` is not).

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. If no return type or argument types are given, the type should be handled as a callable `void$` and assumed to take any amount of arguments with any types.
	
	Syntax: `func[:<return type>([argument types])]`
	
	Examples:
	`func:int(int, char$)`
	`func:int(char$, ...)`

`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:

All program flow keywords are only valid inside of functions unless specified otherwise.

Program Flow		
`break`	`break;`	Exits the current `do`, `for`, 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. Is only valid in the global scope or inside another namespace. The name `global` is reserved.

`continue`	`continue;`	Skips the remaining code in a `do`, `for`, or `while` block.
`defer`	`defer <statement or statement block>`	Adds to the end of the code to be run before exiting the current scope.
`defer:clear`	`defer:clear;`	Removes all the code to be run before exiting the current scope.
`defer:top`	`defer:front <statement or statement block>`	Puts the statements before the rest of the code to be run before exiting the current scope.
`defer:undo`	`defer:undo;`	Undoes the last action performed on the defer code.
`do`	`do [(<condition>)] <statement block>`	Runs code then loops as long as the condition is true and loops forever if no condition is provided.
`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, falls through to `default` if there is no case for that value, and falls through to the case directly under if no case is provided.

`for`	`for (<variable declarations>; <condition>; <each-time statement or block>) <statement 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.
`global`	`global.<name>`	Provides a shortcut to the global scope for use in namespaces. Is valid in all scopes.
`goto`	`goto <name>;`	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.
`label`	`label <name>;`	Creates a label in the current scope with the name given.
`return`	`return <data>;`	Returns from a function and returns data if not `void`.

`switch`	`switch (<variable>) {`
	`  case (<value>) {`
	`  [statements]`
	`  }`
	`  case (<val 1>, <val 2>) {`
	`  [statements]`
	`  }`
	`  case (<value>..<value>) {`
	`  [statements]`
	`  }`
	`  case (<value>...<value>) {`
	`  [statements]`
	`  }`
	`  default {`
	`  [statements]`
	`  }`
	`}`
		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. To have one case for multiple values, separate the values with commas. Placing two dots between two values will match that case for the first value and any values in-between. Placing 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 block>`	Loops code as long as the condition is true.

Assignment		
`cast`	`cast:<type>(<data>)`	Returns data as a different type.

Data		

`<cast from>:cast:<cast to>`	`<type to cast from>:cast:<type to cast to> (<input>)`
	`<statement block>`
		Defines code to perform a cast from one type to another. If no definite `return` is provided, then the cast is considered invalid. The input argument has the types of the first types and the return statement expects to return a value compatible with the second types (the compiler will attempt to use the code for all the combinations possible with the given types).
		
		Examples:
		`my_t:cast:bool(in) {`
		`  return in.valid;`
		`}`
		
		`my_t:cast:int (in) {`
		`  return in.data;`
		`}`
		
		`int:cast:my_t (in) {`
		`  return {true, in};`
		`}`

`enum`	`enum[:<type>] [(<equation>)] {`
	`  <name>[ = <rhs value>];`
	`} [<name>];`
	
	`enum(<name>) <variable declaration(s) or definition(s)>`
		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. If a type is specified after a colon, that type will be used instead of `int`. Only built-in types are allowed. If a name is provided, the enums are given that type with the `enum` wrapper.

`newtype`	`newtype <name> <type>`	Creates a new type using an existing type.
`sizeof`	`sizeof(<type or data>)`	Returns the size in bytes of a type or the type of some data.

`struct`
`structure`
	`struct [modifiers] [<name>] {`
	`  <type>[:<bits>] <name>;`
	`};`
	
	`struct(<name>) <variable declaration(s) or definition(s)>`
		Creates a structure definition or variable. If a colon and number is provided a type, it will be limited to that size in bits and the compiler will attempt to pack adjacent values in the unused bits. If defined inside of a union or another struct, the name is not required and if not provided, variables can be accessed as if they were part of the parent struct or union.

`<type>` `:default`	`<type>:default = <value>`	Sets the default value for a type in the current scope. Only static values are allowed.

`<type>:<op>`	`<type>:<operation> (<var 1>, <var 2>) <statement block>`	Defines code to perform an action on a type. If no definite `return` is provided, then the operation is considered invalid. The return statement expects to return a value of the type given.
		
		Operations: `add`, `and`, `dec`, `div`, `inc`, `lrot`, `lshift`, `mod`, `mul`, `or`, `rrot`, `rshift`, and `xor`.
		
		Example:
		`my_t:add (op1, op2) {`
		`  return op1.v + op2.v;`
		`}`

`typeof`	`typeof(<data>)`	Returns the type of the given data.

`union`	`union [modifiers] [<name>] {`
	`  <variable declarations>`
	`};`
	
	`union(<name>) <variable declaration(s) or definition(s)>`
		Creates a union definition or variable. Each value is stored at the same location and the size will be the size of the largest type at minimum. If defined inside of a struct or another union, the name is not required and if not provided, variables can be accessed as if they were part of the parent struct or union.

Miscellaneous		

asm	asm[:<syntax>] (<quoted register name> <variable name to set register to>) {
	  <assembly>
	}
	
	asm[:<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.

vararg	vararg(<type>) vararg(<varargs_t>, <type>)	Returns the next argument in a variadic function. The single-argument version is invalid in functions that are not variadic. The dual-argument version expects a varargs_t returned by varargs() and is invalid for variadic functions.
varargs	varargs()	Returns a varargs_t that points to the head of the variadic argument data. Is invalid in functions that are not variadic.
varargct	varargct() varargct(<varargs_t>)	Returns the number of variadic arguments passed to the function. The version that does not accept arguments is invalid in functions that are not variadic. The single-argument version expects a varargs_t returned by varargs() and is invalid for variadic functions.

Modifiers		
align	align(<value>) <function declaration and/or definition> align(<value>) <variable declaration(s) and/or definition(s)> {struct|union} align(<value>) <struct or union definition> align(<value>) <statement block>	Align and pad a function, variable, struct, union, or statement(s) to a certain amount of bytes.

`default`	`default <function declaration and/or definition>`	Declares a variant of a function to be the default and making its symbol unmangled. This is only valid on functions. It is required to use `default` on one variant of an overloaded function.
`dynamic`	`dynamic <function declaration and/or definition>`	Allows a function's address to be changed like a function pointer. This is only valid on functions.
`extern` `external`	`extern <variable declaration(s) and/or definition(s)>`	Declares a variable that is stored outside the current file and is invalid on functions and function arguments.
`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. The compiler will embed whenever it can unless it goes against optimization (such as embedding large functions when optimizing for size). This is only valid on functions.
`lang`	`lang:<name> <function declaration and/or definition>`	Tells the compiler that a function uses or should use the ABI of the specified language.
`mangle`	`mangle <function declaration and/or definition>`	Forces a function's symbol to be mangled or generates a mangled symbol if used with `default`.
`nodiscard`	`nodiscard <function declaration and/or definition>`	Generates a warning if the return value is ignored. This is only valid on functions.
`noinit`	`noinit <variable declaration(s) and/or definition(s)>`	Behaves as if the `initvar` pragma is disabled for that specific variable. Is only valid for variables and is invalid for function arguments.

`nooptimize`	`nooptimize <function declaration and/or definition>`
	
	`nooptimize <variable declaration(s) and/or definition(s)>`
	
	`nooptimize <statement block>`
		Makes a function, variable, or statements ineligible for optimization.

`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. Is invalid on function arguments

`reg`
`register`
	`reg[(<register>)] <variable declaration(s) and/or definition(s)>`	Recommends that the compiler use a register to hold the data and is invalid on functions. If a register is supplied, this will force the compiler to use a register and use the specified register to store the variable. A compiler error will be generated if the compiler fails to reserve the register requested.
		
		Example:
		`reg int myvar = 0;`
		`reg(eax) int myvar2 = 1;`
		
		`myvar` might not be held in a register while `myvar2` is guaranteed to be held in `eax`.

`static`	`static <variable declaration(s) and/or definition(s)>`	Makes a local variable hold its value after exiting its scope and is invalid on functions.

Compiler

Compiler requirements:

- The ability to compile multiple files
- The ability to change the output target
- The ability to skip the linking stage and output object files
- The ability to output a linked executable
- Order-independent linking
- A way to define macros outside of source files (preferably using a command-line switch)
- A way to output a list of files included in the provided source files with the ability to change whether system headers are included in the output and change whether relative or absolute paths are used (preferably using a command-line switch)
- A way to add directories to search for includes outside of source files (preferably using a command-line switch)
- A way to add directories to search for libraries outside of source files (preferably using a command-line switch)
- A way to output a list of default compiler macros and the values of those macros (preferably using a command-line switch)
- A way to output a list of macros defined in the provided source files (preferably using a command-line switch)

Preprocessor commands and functions:

In order for a line to be recognized as a preprocessor command, the first non-whitespace character must be 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 otherwise.
The compiler must support at least 10 nested comments.
Comments started with `#<` must end with `#>` and comments started with `/*` must end with `*/`.

`#` `//`	`# <text>` `// <text>`	Comment until end of line.

`#<`
`/*`
	`#< <text>`
	`<text>`
	`<text> #>`
	
	`#< <text> #>`
	
	`/* <text>`
	`<text>`
	`<text> */`
	
	`/* <text> */`
		Start a comment.

`#>`
`*/`
	`#< <text>`
	`<text>`
	`<text> #>`
	
	`#< <text> #>`
	
	`/* <text>`
	`<text>`
	`<text> */`
	
	`/* <text> */`
		End a comment.

`error`	`@error "<string>"`	Raises a compiler error using the provided string as or in the message.
`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 statements 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. If given a directory, all .wh files in the first level will be included. [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. If given a directory, all .wh files in the first level will be included. "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.
warning	@warning "<string>"	Raises a compiler warning using the provided string as or in the message.

Pragmas:

Compiler pragmas only have effect on the current file unless stated otherwise. Meaning that for example, unless stated otherwise in the command description, pragma commands in an included file will not effect the file it was included from unless specified otherwise.

`lazybool`	`lazybool <true/yes, false/no, or no value>`	Causes `bool` to not guard against becoming greater than 1 or less than 0. This is false by default. Providing no value will assume `true`.
`smallbool`	`smallbool <true/yes, false/no, or no value>`	Causes `bool` become the same size as char instead of the same size of int. This is false by default. Providing no value will assume `true`.
`var variable`	`var <comma separated list of <var = val>>`	Sets internal compiler variables.
`initvar`	`initvar <true/yes, false/no, or no value>`	Causes local variables to be set to the default value for their type instead of uninitialized when no initial value is given. This is true by default. Providing no value will assume `true`.
`...`	`...`	Compiler-specific commands can be added.

Macros:

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

Macros take precedence over functions and variables. For example, defining the variable `TEST` and defining a macro `TEST` in the same program will use the macro instead of the variable. It is recommended that the compiler throw a warning about conflicts like this.

`__ABI_<name>`	Names of the language ABIs the compiler supports. There name should match the official capitalization of the language and if the name contains capital letters, a second macro should be defined where the name is lowercase. If the language has any special characters, it should be replaced with the name of that character.
	
	Examples:
	`__ABI_C`
	`__ABI_c`
	or
	`__ABI_CPlusPlus`
	`__ABI_cplusplus`
	or
	`__ABI_CSharp`
	`__ABI_csharp`
	or
	`__ABI_MyLang`
	`__ABI_mylang`

`__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. The name should be uppercase.
	
	Examples:
	`__ARCH_X86_64`
	`__ARCH_AMD64`
	or
	`__ARCH_X86`
	`__ARCH_I386`
	or
	`__ARCH_ARM`
	`__ARCH_ARM32`
	`__ARCH_AARCH32`

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

`__ASM_HAS_<name>`	Names of the assembly syntaxes or assembly compilers 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. The name should be uppercase.
	
	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: `1002010`
`__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 macro containing the name of the current file contained in a string. Example: `"hello.wiz"`
`__FILE_PATH`	A macro containing the file path given to the compiler. Example: `"src/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"`
`__LINE`	A dynamic macro containing the line in the file that this macro was used. Example: `47`

`__NAMESPACE`	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: `"funcs"`
`__NAMESPACE_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. The name should match the type name in capitalization. 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. The name should be uppercase.
	
	Examples:
	`__OS_LINUX`
	`__OS_UNIX`
	or
	`__OS_BSD`
	`__OS_FREEBSD`
	`__OS_UNIX`
	or
	`__OS_WINDOWS`
	`__OS_WINDOWS_NT`
	or
	`__OS_WINDOWS`
	`__OS_WINDOWS_9X`

`__STANDARD <version>`	Version number of the Wizardry standard the compiler supports. Example: `2023051200`
`true (1)`	A value that evaluates as true.

Grammar

Grammar syntax:

- `*:` : Root definition
- `:` : Make definition
- `<>`: Insert definition
- `()` : Group
- `[]` : Optional group or definition
- `{}` : Choice
- `\\` : 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

`*:`

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 under `wizardry/*.wh`.

Any variables, enums, functions, types, or defines in the standard library that are not part of the specification should start with a single underscore.

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

All standard library functions should guard against `NULL` unless specified otherwise.

Headers:

wizardry/	
Variables	
Enumerators	
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`	Input stream file descriptor.
`int wiz.outfd`	Output stream file descriptor.
`int wiz.errfd`	Error stream file descriptor.
Functions	

`int wiz.close(int fd)`	Close a file stream using a descriptor. Returns `0` on success and a negative error code on failure. `fd`: File descriptor to close.
`int wiz.fclose(wiz.file$ file)`	Close a file structure by closing the stream with `close` and deallocating the structure using `free`. Returns `0` on success and a negative error code on failure. `file`: File structure to close.
`int wiz.fdprint(int fd, char$ text)`	Writes a string to a file descriptor. Returns the a negative error code on failure to write all bytes and the number of bytes written otherwise. `fd`: File descriptor. `text`: String to write.
`int wiz.fdprintc(int fd, char chr)`	Writes a char to a file descriptor. Returns `0` on success and a negative error code on failure. `fd`: File descriptor. `chr`: String to write.
`int wiz.fdprintf(int fd, char$ text, ...)`	Writes a formatted string to a file descriptor. Returns the a negative error code on failure to write all bytes and the number of bytes written otherwise. `fd`: File descriptor. `text`: String to write. `...`: Variadic arguments for format.
`int wiz.flush(wiz.file$ file)`	
`wiz.file$ wiz.fopen(char$ path, char$ mode, int$ err)`	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 `err` to an error code if not `NULL`. `path`: File path. `mode`: File mode string. `error`: Pointer to output error code.
`int wiz.fprint(wiz.file$ file, char$ text)`	Writes a string to a file type. Returns the a negative error code on failure to write all bytes and the number of bytes written otherwise. `file`: File type `text`: String to write.

`int wiz.fprintc(wiz.file$ file, char char)`	Writes a char to a file type. Returns `0` on success and a negative error code on failure. `file`: File type `chr`: String to write.
`int wiz.fprintf(wiz.file$ file, char$ text, ...)`	Writes a formatted string to a file type. Returns the a negative error code on failure to write all bytes and the number of bytes written otherwise. `file`: File type `text`: String to write. `...`: Variadic arguments for format.
`int wiz.open(char$ path, char$ 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 a negative error code on failure. `path`: File path. `mode`: File mode string.
`int wiz.print(char$ text)`	Writes a string to the output file stream. Returns the a negative error code on failure to write all bytes and the number of bytes written otherwise. `text`: String to write.
`int wiz.printc(char chr)`	Writes a char to the output file stream. Returns `0` on success and a negative error code on failure. `chr`: String to write.
`int wiz.printf(char$ text, ...)`	Writes a formatted string to the output file stream. Returns the a negative error code on failure to write all bytes and the number of bytes written otherwise. `text`: String to write. `...`: Variadic arguments for format.
Types	
`wiz.file`	Implementation-specific.
Macros	
`wiz.IN_FILENO`	Input stream file number.
`wiz.OUT_FILENO`	Output stream file number.
`wiz.ERR_FILENO`	Error stream file number.

wizardry/dynlib.wh	
Functions	
`void wiz.freelib(wiz.dynlib$ lib)`	Closes and frees a dynamic library. `lib`: Pointer returned by `loadlib`.
`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 success and a negative error code on failure. `lib`: Pointer returned by `loadlib`. `name`: Symbol name
`void$ wiz.getsym(wiz.dynlib$ lib, char$ name, int$ err)`	Returns the pointer to a symbol from a library. Sets `err` to an error code if not `NULL` and returns `NULL` if not successful. `lib`: Pointer returned by `loadlib`. `name`: Symbol name
`wiz.dynlib$ wiz.loadlib(char$ path, int options, int$ err)`	Opens a dynamic library and returns a pointer to be used with other functions. Searches system library directories by default (`/lib` and/or `/usr/lib` on Unix-based operating systems for example). Sets `err` to an error code if not `NULL` and returns `NULL` if not successful. `path`: Path or filename of the library file. `options`: Option flags for finding or loading the library. Combine flags with the bitwise OR operator.
`char$ wiz.mangle(char$ ret, char$ name, char$$ args, int$ err)`	Produces a mangled symbol name from info about a function. The return value should be freed by the caller. `NULL` is returned on failure and `err` is set if not `NULL`.
`int wiz.mangleto(char$ out, luint size, char$ ret, char$ name, char$$ args)`	Writes a mangled symbol name to a string using info about a function. On failure, a negative error code is returned and `out` is left unmodified. On success, the length of the string not including the terminator is returned.
Types	
`wiz.dynlib`	Implementation-specific.
Macros	

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

`wiz.DYN_SEARCH_LOCAL`	Search in the local directory instead of the system library paths.
`wiz.DYN_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.DYN_...`	Other operating system or architecture specific flags can be added.

wizardry/memory.wh	
Functions	
`void$ wiz.alloc(luint size)`	Dynamically allocate a block of memory on the heap. Returns a valid pointer on success and on failure, returns `NULL` if not successful. `size`: Size in bytes of how large the block should be.
`luint wiz.asize(luint size)`	Align a size to the size of a machine word. `size`: Size to align.
`void wiz.copymem(void$ src, void$ dest, luint len)`	Copies a certain amount of bytes from a memory pointer to another without overwrite errors. This function should not guard against `NULL`. `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, luint len, u8 value)`	Writes a certain amount of bytes at a memory pointer. This function should not guard against `NULL`. `ptr`: Pointer to memory to start writing at. `len`: Amount of bytes to write. `value`: Value to write.
`void wiz.free(void$ ptr)`	Free a dynamically allocated block of memory returned by `alloc`. `ptr`: Pointer to the block of memory to be freed.
`void$ wiz.realloc(void$ ptr, luint size)`	Reallocates or resizes a dynamically allocated block of memory returned by `alloc`. Returns a valid pointer on success and `NULL` on failure. Should behave as `alloc` if given `NULL`. `ptr`: Pointer to existing memory block. `size`: New size.

`private inline luint wiz.iasize(luint size)`	Align a size to the size of a machine word. This function should be written in the header. `size`: Size to align.
`private inline void wiz.icopymem(void$ src, void$ dest, luint len)`	Copies a certain amount of bytes from a memory pointer to another without overwrite errors. This function should not guard against `NULL`. This function should be written in the header. `src`: Pointer to memory to reading from. `dest`: Pointer to memory to start writing at. `len`: Amount of bytes to copy.
`private inline void wiz.ifillmem(void$ ptr, luint len, u8 value)`	Writes a certain amount of bytes at a memory pointer. This function should not guard against `NULL`. This function should be written in the header. `ptr`: Pointer to memory to start writing at. `len`: Amount of bytes to write. `value`: Value to write.
`private inline void wiz.izeromem(void$ ptr, luint len)`	Writes a certain amount of zeros at a memory pointer. This function should not guard against `NULL`. This function should be written in the header. `ptr`: Pointer to memory to start writing at. `len`: Amount of zeros to write.
`void$ wiz.valloc(luint 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(luint 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, luint len)`	Writes a certain amount of zeros at a memory pointer. This function should not guard against `NULL`. `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.REALLOC_SAFE_INVAL`	Defined if the `realloc` implementation is protected against invalid pointers.

`wizardry/exit.wh`	
Variables	
`func:void(int) wiz.atexit = NULL`	If non-`NULL`, `wiz.exit` will call this before cleaning up and exiting. The `int` argument passed to the function is the exit code.
Functions	
`default void wiz.exit(int code)` `void wiz.exit()`	Cleans up and exits the program with a code. The second definition will assume the return code to be 0.
Macros	
`wiz.EXIT_SUCCESS`	A macro containing the success return code.
`wiz.EXIT_FAILURE`	A macro containing the failure return code.

`wizardry/error.h`	
Enumerators	
`wiz.ERR_NONE = 0`	An enum error code for no error.
`wiz.ERR_2BIG`	An enum error code for too many arguments.
`wiz.ERR_ALLOC`	An enum error code for a memory allocation error.
`wiz.ERR_ARG`	An enum error code for an invalid argument.
`wiz.ERR_BADFD`	An enum error code for an invalid file descriptor.
`wiz.ERR_BUSY`	An enum error code for a busy device or resource.
`wiz.ERR_CANCELED`	An enum error code for a cancelled operation.
`wiz.ERR_DQUOT`	An enum error code for an exceeded disk quota.
`wiz.ERR_EXIST`	An enum error code for a path already existing.
`wiz.ERR_FBIG`	An enum error code for a too large file.
`wiz.ERR_INVAL`	An enum error code for invalid data.
`wiz.ERR_ISDIR`	An enum error code for an unexpected directory.
`wiz.ERR_ISFILE`	An enum error code for an unexpected file.
`wiz.ERR_LINK`	An enum error code for too many symbolic link levels.
`wiz.ERR_MEM`	An enum error code for a memory allocation error.
`wiz.ERR_MFILE`	An enum error code for exceeding the open file limit.
`wiz.ERR_N2LONG`	An enum error code for a too long name.

`wiz.ERR_NOEXIST`	An enum error code for a nonexistent path.
`wiz.ERR_NOTEMPTY`	An enum error code for a non-empty directory.
`wiz.ERR_NULL`	An enum error code for an unexpected `NULL`.
`wiz.ERR_OVERFLOW`	An enum error code for a data overflow.
`wiz.ERR_PERM`	An enum error code for permission denied.
Functions	
`char$ wiz.errstr(int code)`	Returns a string associated with an error code. `code`: Error code to get string for.

`wizardry/string.h`	
Variables	
Enumerators	
Functions	
`private inline bool wiz.istrcmp(char$ str1, char$ str2)`	
`private inline bool wiz.istrcasecmp(char$ str1, char$ str2)`	
`fastbool wiz.strcmp(char$ str1, char$ str2)`	
`fastbool wiz.strcasecmp(char$ str1, char$ str2)`	
`char$ wiz.strdup(char$ str)`	
`int wiz.strcat(char$ dest, luint len, char$ src)`	
Types	
Macros	

Formatted text:

t

OS-Specific Library

All OS-specific functions, variables, and type definitions should be held in the `os` namespace.

OS-specific headers should be split into separate headers under `os/*.wh`.