Ether programming language

Sergey Basalaev

June 7, 2013

Abstract

This document is a brief description of Ether programming language for Alchemy OS.

Contents

1 Building a program

Let's start with the usual example – "Hello world" in Ether. To try it create file `hello.e` in the `/home` directory with the following contents:

```
/* EXAMPLE: Hello world */

use "io"

def main(args: [String]) {
  println("Hello, world!")
}
```

To convert this program in binary executable format you need to compile it. This is done with `ex`, Ether compiler for Alchemy OS. Open *Terminal* and execute the following command:

```
ex hello.e -o hello
```

This command will produce executable file named `hello`. To run this file type in terminal:

```
./hello
```

2 Program structure

Ether sources are written in UTF-8 encoding. Newlines are not meaningful.

2.1 Comments

Comments are completely ignored by compiler.

```
// This comment spans till the end of the line

/*
This is a block comment.
It can span several lines.
*/
```

2.2 Functions

Ether program is a set of functions. Function is defined using one of the following declarations:

```
// this is a function called 'name'
def name(arg1: Type1, ..., argN: TypeN): ReturnType {
  expr1
  expr2
  ...
  lastexpr
}

// this is a function without return type (procedure)
def proc(arg1: Type1, ..., argN: TypeN) {
  expr1
  expr2
  ...
  lastexpr
}
```

When function is invoked, its expressions are evaluated sequentially. If return type is specified, then result of the last expression is used as return value. If return type is not specified, function does not return a value.

Simple functions can also be defined using “function=expr” syntax like:

```
// returns maximum of two numbers
def max(a: Int, b: Int): Int = if (a>b) a else b
```

2.3 Main function

Each program must have a function called ”main”. This function is invoked when program starts and accepts command-line arguments as array of strings.

```
def main(args: [String]): Int {
  ...
}

// can be also defined as procedure
def main(args: [String]) {
  ...
}
```

The ”main” function returns ”program exit code”. Usually, zero exit code means that program ended normally and non-zero exit code indicates an error. If ”main” is declared as procedure, program always ends with zero exit code.

2.4 Includes

There are many functions provided by standard Alchemy libraries. To use one of them you need to include corresponding header file in your program. This is done using the following syntax:

```
use "header_name"
```

Standard header files are located in `/inc` directory (take a look at them). To learn more about these headers and functions they contain proceed to API reference.

3 Variables

3.1 Declaring variables

Variables are declared using `var` keyword. In the next example program asks user for input, stores entered text in a variable and then uses that variable in output.

```
/* EXAMPLE: Asking user for input. */

use "io"

def main(args: [String]) {
  println("Enter your name:")
  var name = readline()
  println("Hello, " + name + "!")
}
```

Ether has strong typing and every variable is of certain type. Type of variable is determined from expression that is assigned to it. In some cases you might want to specify type manually. This can be done using the following syntax:

```
// declaring string variable
var str: String

// declaring and assigning in one line
var a: Any = 3.14
```

Name of variable may include letters, numbers and underscore characters (`'_'`). Names are case sensitive, i.e. `doo`, `Doo` and `DOO` are three different variables. The following words are reserved and can't be used as variable names.

```
cast    catch   const   def
do      else    false   for
if      new     null    super
switch  this    true    try
type    use     var     while
```

3.2 Local and global variables

Variable declared in a function is *local*, i.e. it is visible and can be used only inside that function. If variable is defined in a block {...} then it is visible

only inside that block. You may also define variables at outer level making them *global*.

Note: local variables always work faster than global.

3.3 Constants

Constant is a variable value of which cannot be changed after creation. Constants are created with the same syntax as variables, but using the keyword `const`. Constants can also be declared on outer level, in this case value assigned to them must be a literal expression (number, string or boolean value). Note, that compiler is smart enough to evaluate constant from simple expression, so you can safely use something like

```
const DEBUG = true

const WORD = if (DEBUG) "Debug" else "Release"

def main(args: [String]) {
  println("This is " + WORD + " build")
}
```

4 Types

Ether has hierarchical type system. The most basic type is `Any`. All others are its subtypes. Subtypes inherit all properties and methods from parent types but may define their own.

```
/* Builtin types */

Any
 -numeric types
   -Byte
   -Short
   -Char
   -Int
   -Long
   -Float
   -Double
 -Array
   -all array types
 -Structure
```

```
  -all structure types
-Function
  -all function types
-String
-Error
```

Type `Any` represents value of any type. It is used generally when the type of the value is unknown.

4.1 Numeric types

Values of numeric types are all kinds of numbers. Numbers can part in arithmetic expressions.

Type	Description
`Byte`	Single byte as signed 8-bit integer in range from -128 to 127.
`Short`	Signed 16-bit integer in range from -32767 to 32768.
`Char`	Single unicode character as number in range from 0 to 65535.
`Int`	Signed 32-bit integer number in range from -2^{31} to $2^{31}-1$.
`Long`	Signed 64-bit integer numbers between -2^{63} and $2^{63}-1$, inclusive.
`Float`	Single precision (32-bit) floating point number.
`Double`	Double precision (64-bit) floating point number.

4.2 Bool type

Type `Bool` represents boolean logical values `true` and `false`. Boolean values are used in logical expressions and conditional operators.

4.3 String type

Strings represent text information as a sequence of characters. Strings in Ether are immutable. Any value can be converted to a string (e. g. for printing) using `tostr()` method.

4.4 Arrays

Array is a fixed length sequence of elements of the same type. Array type is defined as element type enclosed in square brackets (for example, `[Int]`). There are three constructs that create new array.

(1) The following code creates array of 10 `String` elements. Elements of array are initially uninitialized (are equal to `null`).

```
var array = new [String](10)
```

(2) Array can be created by listing all its elements. Length of array is a number of elements in list. Type of array is determined from types of elements.

```
// This array is [Int]
var numbers = [1, 2, 3, 4, 5]
// This array is [Any] because it contains values of different types
var things = [3.14, "Pi"]
// Zero-length array is also [Any]
var zerolength = []
```

(3) Finally you can specify both type of array and its elements.

```
// will be [1.0, 2.0, 3.0]
var dbl = new [Double] { 1, 2, 3 }
```

The length of the array can be obtained using `array.len`. Array elements are indexed from `0` to `len-1`. To access i-th element of array syntax `array[i]` is used.

```
/* EXAMPLE: work with arrays. */
use "io"

def main(args: [String]) {
  // getting length of array
  println("Number of arguments: " + args.len)
  // printing all elements
  for (var i=0, i<args.len, i+=1) {
    println("Argument #" + i + ": " + args[i])
  }
}
```

4.5 Structures

Structure is a composite type that contains a set of values. Structure types are defined as follows:

```
type Struct {
  field1: Type1,
  field2: Type2,
  ...
  fieldN: TypeN
}
```

Structures are created using *constructors*. If no self-defined constructor provided (see constructors), two default constructors are provided:

```
// compact constructor
var s = new Struct(val1, val2, ..., valN)

// extended constructor
var t = new Struct {
  field1 = val1,
  field2 = val2,
  ...
  fieldN = valN
}
```

Compact constructor lists values of all fields in order of declaration. Field values in extended constructor may follow in any order and may be skipped. Skipped fields are initially unset (their value is `null`) or set to default values. You may provide default values to fields of basic types (numbers, strings, characters and booleans).

```
type Person {
  name: String = "John",
  age: Int = 20
}
```

Structure fields are accessed using dot.

```
/* EXAMPLE: Declaring structure. */
use "io"

type Complex {
  re: Double,
  im: Double
}

def main(args: [String]) {
  var z = new Complex(-1.5, 4)
  println("z = " + z.re + "+" + z.im + "i")
  // => -1.5+4i
}
```

4.6 Functional types

Functional type is a type of procedure or function. The syntax of the function type is:

```
(Type1, Type2, ..., TypeN): ReturnType
(Type1, Type2, ..., TypeN)
```

If return type is omitted, then type represents procedure.

Functions can part in expressions like any other values. In the next example we create variable of functional type and assign different values to it.

```
/* EXAMPLE: Using function values. */
use "io"

def max(a: Int, b: Int): Int = if (a > b) a else b

def main(args: [String]) {
  var f: (Int,Int):Int
  f = max
  println(f(3,5)) // => 5
  f = def(a:Int, b:Int): Int {a + b}
  println(f(3,5)) // => 8
}
```

5 Expressions

5.1 Constants

null constant

The keyword `null` represents special value that is used to indicate that some object is uninitialized. New variables, elements of new arrays, fields of new structures that were not set explicitely are set to `null`.

There are only three operations that end normally using `null` value:

- Testing whether object is `null` or not: `val == null`
- Assigning `null` value to object. If variable is set to `null`, memory occupied by it is freed and it enters uninitialized state.
- Calling `null.tostr()` returns string `"null"`.

Attempts to perform other operations on uninitialized object end with `ERR_NULL` error.

Integer numbers

Integer numbers are values of types `Int` or `Long`. Ether recognizes decimal numbers and hexadecimal numbers preceded with `0x`. By default, numbers are of type `Int`. To write `Long` number you need to add `L` or `l` to it. Examples:

```
42    0x1CCF    123L
```

Floating point numbers

Floating point types are `Float` and `Double`. Ether recognizes numbers with decimal dot (`3.14`, `0.`, `.15`) and numbers in exponential form (`1e15`, `31.4e-1`, `.15e+4` which represent $1 \cdot 10^{15}$, $31.4 \cdot 10^{-1}$, $0.15 \cdot 10^{4}$ respectively). By default, numbers are of type `Double`. To indicate that number is `Float` you need to add suffix `F` or `f`. To indicate that number is `Double` you may add suffix `D` or `d`. For instance, `5d` is `Double` and `.15e+4F` is `Float`.

Boolean values

Boolean values are represented by keywords `true` and `false`.

Strings

String literal is a sequence of zero or more characters and escape sequences enclosed in double quotes. Examples:

```
""        "This is a string"        "Pi symbol is \u03C0\n"
```

The following escape sequences are supported:

`\n`	new line;
`\t`	horizontal tabulation;
`\r`	return;
`\b`	backspace;
`\f`	form feed;
`\'`	single quote (`'`);
`\"`	double quote (`"`);
`\\`	backslash (\);
`\nnn`	ASCII character by its octal code, `nnn` is octal number in range `0..377`;
`\uXXXX`	Unicode character by its hexadecimal code, `XXXX` is exactly four hexadecimal digits.

Characters

A character in single quotes is a `Char` value. Escape sequences are also supported.

```
'a'    '\n'    '\u03C0'
```

5.2 Operators

Arithmetics

Arguments of arithmetic operators are numbers (`Int`, `Long`, `Float` or `Double`). If arguments are numbers of different types, result is the most wide type.

`- a`	Negation
`a + b`	Addition
`a - b`	Subtraction
`a * b`	Multiplication
`a / b`	Division
`a % b`	Remainder from division

Equality

Equality operators can be applied to arguments of any type. Result is either `true` or `false`.

`a == b`	Equals
`a != b`	Not equals

Number comparison

Arguments of arithmetic operators are numbers (`Int`, `Long`, `Float` or `Double`). If arguments are numbers of different types, they are converted to the most wide type before comparison.

`a < b`	Less than
`a <= b`	Less than or equal to
`a > b`	Greater than
`a >= b`	Greater than or equal to

Logical operators

Logical operators are operators between `Bool` values. Result is either `true` or `false`.

`! a`	Logical NOT
`a & b`	Logical AND
`a && b`	Lazy logical AND
`a | b`	Logical OR
`a || b`	Lazy logical OR
`a ^ b`	Exclusive OR

Bitwise operators

Bitwise operators perform logical operations between bits of integer numbers. Both arguments must be `Int` or `Long`.

`~ a`	Bitwise negation
`a & b`	Bitwise AND
`a | b`	Bitwise inclusive OR
`a ^ b`	Bitwise exclusive OR

Bit shifts

Operators of this group shift bits of given integer number. The first argument is either `Int` or `Long`. The second argument specifies amount of bits to shift and must be an integer in range `1..31` for `Int` or `1..63` for `Long`.

`a << b`	Shift bits left
`a >> b`	Shift bits right saving sign bit
`a >>> b`	Shift bits right

String operators

`str1 + str2`	Concatenates two strings together. If second argument is not a string, it is converted to a string using `Any.tostr()`.
`str[at]`	Returns character at the specified position of the string.
`str[from:to]`	Returns sequence of characters in positions *from* .. *to*-1.
`str[:to]`	Is equal to `str[0:to]`.
`str[from:]`	Is equal to `str[from:str.len()]`.

Function invocation ()

Postfix operator `expr(arg1,arg2,...)` executes function defined by expression `expr`. The expression must be of functional type.

Operator precedence

The following table shows order in which operators are applied. The higher category is, the earlier operator will apply. Operators from the same category are applied from left to right (right to left for prefix operators).

Category	Operators
Postfix expr`[at]` expr`[from : to]`	expr`(...)`
Prefix `-`expr `+`expr `~`expr	`!`expr
Multiplicative	`* / %`
Additive	`+ -`
Shifts	`<< >> >>>`
Equality	`== !=`
Comparison	`< <= > >=`
AND	`& &&`
Inclusive OR	`| ||`
Exclusive OR	`^`

5.3 Type cast

Syntax:

```
expr.cast(Type)
```

Type cast is used in cases when actual type of expression cannot be predicted by compiler. Also, it can be used to convert between numeric types.

5.4 Block expression

Syntax:

```
{
  expr1;
  expr2;
  ...
  exprN;
}
```

Block expression is a sequence of zero or more expressions enclosed in curly braces. Result of block is the result of the last expression. If last expression does not return value, then the whole block does not return value. Empty block {} may be used as empty expression. Semicolons are optional and may be omitted if that does not lead code to ambiguity.

Variables defined within block are visible only within this block.

5.5 Conditional expression

Syntax:

```
if (condition) expr1 else expr2
if (condition) expr
```

The `condition` must return `Bool` value. If it is `true`, `expr1` is evaluated, otherwise `expr2` is evaluated. The result of this operator is the value of evaluated expression.

```
// Returns maximum of two numbers
def max(a: Int, b: Int): Int = if (a > b) a else b
```

If `else` branch is omitted, then nothing is executed when condition is `false`.

5.6 Switch expression

Syntax:

```
switch (val) {
  a1, a2, ..., aM: expr1
  b1, b2, ..., bN: expr2
  ...
  else: expr_else
}
```

Switch expression chooses one of branches of execution depending on given value. Expressions a_i, b_i must be `Int` constants (or expressions from which constants are easily calculatable) and must differ from each other. Expression `val` must be of type `Int`. If it equals one of numbers a_1, ..., a_M, then `expr1` is evaluated and returned. If it equals one of numbers b_1, ..., b_N, then `expr2` is evaluated and returned, and so on. If number returned by `val` differs from all given constants, then `expr_else` is evaluated and returned. If `else` branch is omitted, then nothing is calculated.

```
// Returns factorial of a number
def fac(n: Int): Long = switch (n) {
  0, 1: 1L
  else: n * fac(n-1)
}
```

5.7 While loops

Loop with precondition:

```
while (condition) expr
```

The `condition` must return `Bool` value. If it is `true` then `expr` is evaluated and condition is checked again.

Loop with postcondition:

```
do expr while (condition)
```

The only difference of this loop from previous is that condition is checked after `expr` is evaluated, so the body of this loop is always executed at least once.

5.8 For loop

Syntax:

```
for (init, condition, increment) expr
```

Before the first cycle of a loop `init` expression is evaluated. Then, `condition` is checked. If it is `true`, then `expr` is evaluated, `increment` is evaluated and `condition` is checked again.

If you want to omit `init` or `increment`, use empty expression {}.

"For" loop is widely used when number of cycles in a loop is known or fixed. For example, it can be used to iterate over array elements:

```
for (var i = 0, i < array.len, i += 1) {
  println(array[i])
}
```

5.9 Anonymous functions

Functions can be defined right in place where they are needed using the following syntax:

```
def(arg1: Type1, ..., argN: TypeN): RetType = expr

def(arg1: Type1, ..., argN: TypeN): RetType {
  expr1;
  expr2;
  ...
}
```

For example:

```
var list = new List()
list.addall(["tar", "top"])
list.mapself( def(a: Any): Any = "s"+a )
println(list) // => [star, stop]
```

5.10 Try/catch operator

When function fails to end normally (for instance, tries to divide by zero), it raises an error. Usually, program just ends and prints error message. Alternatively, error can be catched by try/catch operator.

Syntax:

```
try expr catch (var e) errexpr
try expr catch errexpr
```

If `expr` evaluates normally then its value is returned. Otherwise, error object is stored in variable `e` and `errexpr` is evaluated and returned. If error object is not needed, variable definition may be omitted.

```
try {
  println(a / b)
} catch {
  println("Division by zero")
}
```

6 Special functions

Some functions are treated specially by compiler. They allow to express things easier.

6.1 Methods

Method is a function bound to specific type. Semantically, method is an action which can be performed on a value. For example, output byte stream (type `OStream`) has method `write(byte)` which writes given byte to the stream. To call an object method, value is followed by dot and then by function:

```
// writing 'A' character to the output stream
stdout().write('A')
```

You can define your own methods for new types and even for existing types. Syntax of method is

```
def Type.name(arg1: Type1, ... argN: TypeN): ReturnType {
  function body
}
```

Keyword 'this' is used to refer to the owner of the method. In the following example we add to the complex number type method that converts it into a string.

```
/* EXAMPLE: Defining a method. */

use "io"

type Complex {
  re: Double,
  im: Double
}

def Complex.tostr(): String {
  "" + this.re + "+" + this.im + "i"
}

def main(args: [String]) {
  var c = new Complex(1, 2)
  println(c.tostr())  // => 1.0+2.0i
}
```

Note, that method `tostr()` is commonly used to convert several builtin types to a string. You can read more about it in an API documentation for `Any.tostr()`. In our example, however, method gets overriden and `Complex.tostr()` is used instead. Be warned, that this override is not dynamic, which method to call is resolved statically at compilation time.

```
/* EXAMPLE: inherited type with overriden method. */

type A { }

def A.tostr(): String = "A"

type B < A { }
```

```
def B.tostr(): String = "B"

def main(args: [String]) {
  // variable is of type A
  var b: A;
  // actual value is of type B
  b = new B()
  // but still A.tostr() is used
  println(b.tostr())  // => A
}
```

6.2 Constructors

Constructor is a special function used to allocate and initialize structure. Constructors are defined using the following syntax:

```
def Type.new(arg1: Type1, ..., argN: TypeN) {
  this.field1 = value1
  ...
  // other code
}
```

and then used with "new" keyword.

```
var obj = new Type(arg1, ..., argN)
```

6.3 Array operators

Array operators `[]` and `[]=` can be overloaded using methods

```
def Type.get(at: IndexType): ValueType
def Type.set(at: IndexType, value: ValueType)
```

thus giving a type array like properties.

Expression	Translated to
`value[index] = expr`	`value.set(index, expr)`
`value[index]`	`value.get(index)`

For example, lists and dictionaries define these methods and may be used with such syntax.

If indices used in get/set are `Int` numbers, then range operator `[:]` can also be overloaded. To use it you should define the following methods

```
def Type.range(from: Int, to: Int): Type
def Type.len(): Int
```

This will cover three use cases:

Expression	Translated to
`value[from : to]`	`value.range(from, to)`
`value[: to]`	`value.range(0, to)`
`value[from :]`	`value.range(from, value.len())`

6.4 Properties

To declare a property `foo` of the type use the following methods.

```
def Type.get_foo(): PropertyType
def Type.set_foo(value: PropertyType)
```

Properties use the same syntax as structure fields. If only `get_` is defined, property is read-only. If only `set_` is defined, property is write-only.

Expression	Translated to
`value.foo = expr`	`value.set_foo(expr)`
`value.foo`	`value.get_foo()`

Note, that structure fields take preference over properties with the same name.

6.5 Conversion to a string

String representation of a value is returned by method

```
def Type.tostr(): String
```

If this method is defined, it is called automatically when you use string concatenation and by the following functions

```
print
println
OStream.print
OStream.println
StrBuf.append
StrBuf.insert
```

6.6 Equality operators

Equality operators can be overriden using the following method.

```
def Type.eq(other: Type): Bool
```

Operators are overriden as follows:

Expression	Translated to
`value == other`	`value.eq(other)`
`value != other`	`!value.eq(other)`

This method should obey the general contract of equivalence relation. That is, for any values `x`, `y` and `z` of the type `Type` the following holds:

- `x.eq(x)` always returns `true`;
- `x.eq(y)` and `y.eq(x)` return the same result;
- if `x.eq(y)` is true and `y.eq(z)` is true then `x.eq(z)` is also true.

6.7 Comparison operators

Comparison operators can be overriden using the following method.

```
def Type.cmp(other: Type): Int
```

Operators are overriden as follows:

Expression	Translated to
`value < other`	`value.cmp(other) < 0`
`value > other`	`value.cmp(other) > 0`
`value <= other`	`value.cmp(other) <= 0`
`value >= other`	`value.cmp(other) >= 0`

This method should obey the general contract of order relation. That is, for any values `x`, `y` and `z` of the type `Type` the following holds:

- `x.cmp(x)` always returns 0;
- `x.cmp(y) == -y.cmp(x)`;
- if `x.cmp(y) <= 0` and `y.cmp(z) <= 0` then `x.cmp(z) <= 0`;
- if `x.cmp(y) < 0` and `y.cmp(z) < 0` then `x.cmp(z) < 0`.

6.8 Other operators

Expression	Translated to
`-x`	`x.minus()`
`!x`	`x.not()`
`x + y`	`x.add(y)`
`x - y`	`x.sub(y)`
`x * y`	`x.mul(y)`
`x / y`	`x.div(y)`
`x % y`	`x.mod(y)`
`x & y`	`x.and(y)`
`x | y`	`x.or(y)`
`x ^ y`	`x.xor(y)`
`x << y`	`x.shl(y)`
`x >> y`	`x.shr(y)`
`x >>> y`	`x.ushr(y)`

A Appendix: Compiler options

The following options are supported by `ex`:

`-o` ***name*** Use the following name for output file. By default, `a.out` is used.

`-I`***dir*** Search headers also in this directory. By default headers are searched in the current directory and in `/inc`.

`-O`***level*** Use specified optimization level.

`-O0` Turn all optimizations off.

`-O1` Use basic optimizations. This is the default behavior.

`-O2` Use experimental optimizations.

`-l`***name*** Link executable with library *libname*.

`-L`***dir*** Search libraries to link with also in the following directory. By default libraries are searched in `/lib`.

`-s`***soname*** Add soname to the built library.

`-g` Generate debugging info. With this option information about source file is included in generated binary.

`-c` Only compile sources, but do not link.

`-W`***category*** Turn on specified category of warnings. To turn off warning category use `-Wno-`*category*. You may turn on/off all warnings using `-Wall` or `-Wno-all`.

`-X`***feature*** Turn on experimental feature.

B Appendix: Warning messages

Warnings about unnecessary casts

These warnings indicate that cast() expression is redundant since expression is already of the requested type.

To enable: `-Wcast`
To disable: `-Wno-cast`
Default: enabled

`Unnecessary cast to the same type`

`Unnecessary cast to the supertype`

Warnings about names masking existing names

To enable: `-Whidden`
To disable: `-Wno-hidden`
Default: enabled

`Variable Name hides another variable with the same name`

There is already variable with this name at outer level. It is recommended to rename variable to avoid confusion.

Warnings about main() function

Function called main() is an entry point for program execution. It must receive an array of strings and return Int (or be a procedure). If you are using function for another purpose and call it main() it is better to rename the function.

To enable: `-Wmain`
To disable: `-Wno-main`
Default: enabled

`Incorrect number of arguments in main(), should be ([String])`

`Incompatible argument type in main()`

`Argument of main() should be of type [String]`

`Incompatible return type in main(), should be Int or <none>`

Warnings about overloaded operators

These warnings are shown on methods that violate conventions used in Ether when overloading operators.

To enable: `-Woperators`

To disable: `-Wno-operators`

Default: enabled

`Constructor returns value of different type than Type`

Constructor is a method used to create new instance of specified type. If you are using method named .new() for other purposes then creating new object it is better to rename method.

`Method Type.eq cannot be used as override for equality operators`

Equality override must accept exactly one argument of the same type as owner and return Bool. If you are using method named .eq() for other purposes than equality comparison it is better to rename method.

`Method Type.cmp cannot be used as override for comparison operators`

Comparison override must accept exactly one argument of the same type as owner and return Int. If you are using method named .cmp() for other purposes than order comparison it is better to rename method.

`Method Type.tostr cannot be used as override for Any.tostr()`

Method tostr() is commonly used to build string representation of the object. It must accept no arguments and return String. If you are using method .tostr() for other purposes it is better to rename method.

Warnings about entities in included files

Every source processed by compiler is combined with includes to create *compilation unit*. If some entity is implemented in header or other file included using `use` directive, and this file is included in more than one source, each compilation unit will have its own copy of the entity. E.g., if global variable is defined in header, each source will have its own copy of this variable, i.e. actually distinct variables. Warnings in this group indicate such cases.

To enable: `-Wincluded`

To disable: `-Wno-included`

Default: enabled

`Global variable Name in included file`

Global variables are unique in the compilation units. If this file is included in more than one source, each compilation unit will have its own copy of the variable.

`Function Name is implemented in included file`

Functions are unique in the compilation units. If this file is included in more than one source, each compilation unit will have its own copy of the function.

Warnings about unsafe type casts

Warnings in this group indicate automatic type cast when expected type differs from actual.

To enable: `-Wtypesafe`

To disable: `-Wno-typesafe`

Default: enabled

`Unsafe type cast from Type1 to Type2`

Indicates that expected type differs from that given by expression. If you are completely sure about actual type of expression you can suppress this warning using .cast()

`Unsafe type cast when copying from Array to [Type]`

It is not recommended to use untyped arrays. The code should be rewritten to use arrays of certain types.

`Unsafe type cast when copying from [Type1] to [Type2]`

Indicates that destination array in acopy() contains elements of different type. Therefore, care must be taken to ensure that source array contains only elements of destination type. This warning cannot be suppressed in individual cases.

`Function.curry is not type safe since actual function type is unknown`

Compiler cannot verify safety of using method curry() because type of a function is unknown at the compilation time. It is adviced to use certain functional types instead of general `Function` where it is possible. If you are completely sure about type of a function you can suppress this warning by casting function to the needed type.

Deprecation warnings

Warnings marked with word "deprecated" define things that are not recommended to use and will be removed in subsequent versions.

To enable: `-Wdeprecated`

To disable: `-Wno-deprecated`

Default: enabled