Meucci programming language

Version 0.9.9

De Donato Paolo

Contents

Chapter 1

Introdction

1.1 Overview

Meucci programming language focuses on the concept of **module** instead of Lisp functions or Java objects. A module in Meucci is simply a **container of functions and objects that perform a particular job**. The concept of module is very similar to C++ namespace or Java package, but these languages have focused on objects (especially Java).

Those who had never programmed doesn't understand the above lines. To help them we use this example:

*Suppose that our program is a liquor factory. It's arranged in several **modules**, which are equivalent to factory departments. We consider now distillation department. **Types** could be one of the fermented pomace, one of the distilled pomace, another flavors ... all the real objects which you can "read from" and/or "write to". Certainly doesn't exist one flavor type, there are for example strawberry flavor, lemon flavor, cherry flavor, etc. . Each of this flavors is equivalent inside our program of a **variable of type "flavor"**. **Functions/Operations** may be distillers or any machinery that transforms raw materials (data passed to functions) into processed products (return value). In a big liquor factory doesn't exsist only distillation department, there's also fermentation department, distribution department, contability department, marketing department, ... If the factory is small, it's reasonable to create only one department, but it becomes very ploblematic with increasing of factory dimensions.*

1.2 Compilation

To create an executable from sources it needs a programm called **compiler**. The Meucci programming language compiler is called `mecc`, sources files must have `.x` extension. Now we want to compile source file named `Start.x` that contains:

```
modulo Start{
  int main(){
    println("Stringa stampata a video!");
    //Comment ignored by compiler
  }
}
```

If you want to compile and run it on Linux-based operative systems you go, with your preferred shell, in Start.x folder and type:

```
meucci@Meucci:/home/meucci/Code$mecc Inizio.x
```

The compiler generates the executable `a.out`.

```
meucci@Meucci:/home/meucci/Code$./a.out
Stringa stampata a video!
meucci@Meucci:/home/meucci/Code$
```

Analysis

The source file first line, `modulo Start`, defines a new module called Start (It's not necessary, but hightly recommended, that module name and file name are equal). Module body is enclosed by curly brackets

```
{
...
}
```

In this case the module contains a function called `main` without input parameters returning a number (a particular number type express in Meucci as `int`). This function calls another function

```
println("Stringa stampata a video!");
```

to print a string. main function is a particular function because all programms written in Meucci starts calling this function.

Generated files

Besides a.out, the compiler has generated another file called `Start.in`. This file is similar to .h files in C/C++, in other words allows other modules to use functions and types declared in `Start` module.

This topic will be discussed in modules chapter.

Chapter 2

Fundamental istructions

2.1 Variables declaration

Any program, before being executed by CPU, is copied into **memory** (called also **DRAM**). Memory is seen by CPU as a continuos sequance of bytes, to each of them an adress is assigned. A **variable** in Meucci (and in any other programming language) is only an **adress** of a **memory location** (one or more consecutive bytes) of DRAM, from which you can **read data** or **write data**, that the program use to store informations useful to elaboration.

In order to use a variable it has to be **declared**, i.e. you have to communicate to it its existance so that the compiler connect it to an adress. In a variable declaration it have to be specificated variable **name**, that will be used for accessing it, and data **type** the variable can contains. Variable name can be any sequence of letters or numbers, unless it doesn't contain any space and first character is a letter. Then labels in Meucci are **case sensitive** (it distinguishes uppercase from lowercase). This code line shows a variable declaration:

```
int a;
```

In this code it's declared a `int` type variable with name `a`. In Meuccci at the end of each instruction it has to put `;` character so, because variable declaration is an instruction, it's been ended with `;`.

Now we analyse this more complex case:

```
int a;
int b;
String c;
a = 4;
b = a * 2 + 8;
```

It's been declared three different variables: two `int` and one `String`. To write a data in a variable it uses `=` symbol, with at left variable name in which you

write data, at right that data (in this example `4`). You can put at right of `=` more complex expressions, in fact at last line program does these operations:

1. Reads value stored in `a`
2. Multiplies it by `2`
3. Adds it to `8`
4. Writes result in `b`

At the end of execution `a` variable will contain 4, while `b 16`.

When it declares new variable it's in **unclear state** (there's no way to known its value), so before any use you need to **initialze** it, that is, you need to write in a known value. In Meucci you can declare and initialize variables in one istruction:

```
int a = 4;
int b = a * 2 + 8;
String c;
```

2.1.1 auto declarations

Inside declarations you must specify its type. Type name is often, especially with template parameters, too long. In Meucci you can use **auto** instead of type name, so that compiler determines automatically type of that variable:

```
auto x = 32;
```

Remember that declarations with `auto` **require** initialization, or rather compiler will generate an error.

2.1.2 Variable visibility

An **istruction block** is simply a sequence of istructions between two curly brackets. Correct use of blocks increases code readability and can reduse unused memory with variable **visibility** (or **scope**).

In other words, if a variable inside an istruction block it can be accessed only by istructions inside this block, and the compile can free memory outside that block reusing it.

2.2 Primitive types and constants

In previous example it's been used `int` type as generic number type. In Meucci there are two types class differentiated by how datas are stored in memory: **primitive** types are the simpliest because address associated to variable points directly to data; **reference** types instead contain a **pointer** to data. In this section we point only primitive types.

Primitive types are determined directly by compiler, and are:

- `byte` integer signed number stored in 1 byte (between -128 and 127 included)
- `short` integer signed number stored in 2 bytes (between -32,768 and 32,767)
- `int` integer signed number stored in 4 bytes (between -2,147,483,648 and 2,147,483,647)
- `long` integer signed number stored in 8 bytes (between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807), available only on 64-bit platforms
- `ubyte` integer unsigned number stored in 1 byte (between 0 and 255)
- `ushort` integer unsigned number stored in 2 bytes (between 0 and 65,535)
- `uint` integer unsigned number stored in 4 bytes (between 0 and 4,294,967,295)
- `ulong` integer unsigned number stored in 8 bytes (between 0 and 18,446,744,073,709,551,615), available only on 64-bit platforms
- `char` character in 1 byte (ASCII)
- `real` double precision floating point number (52 bit mantissa, 11 bit exponent)
- `pt` generic pointer (like `void *` of C), dimension depends by architecture
- `boolean` boolean value (true or false) stored in 1 byte

Costants are values determined at compilation time, `2, 4, 8` are constants examples. As variables, constants have a type, whis is determined by these rules:

- An `u` o `U` before integer numeric constant says type is unsigned, otherwise it's signed;
- If numeric constant ended with `b` or `B` then it's `byte` or `ubyte`, if `s` o `S` then `short` or `ushort`, if `l` or `L` then `long` or `ulong`, otherwise `int` or `uint`
- ASCII characters must be enclosed by (')
- Floating point constants must contain only a dot, followed by at least one digit (also 0)

Examples of constants:

```
43
u12S
76b
45.0
U11
```

```
908753129L
'h'
123.45632
```

whose types are respectly: `int`, `ushort`, `byte`, `real`, `uint`, `long`, `char`, `real`. The one `pt` constant is `null`, which is always an illegal address (there are other methods to assign to `pt` variables a valid address).

2.3 Type conversions

Meucci programming language has a strong protection type mechanism: to every variable of a particular type it's possible to assign only constants or variables of the same type: following istructions will generate compilation errors:

```
int a = u4;
```

```
int a = 4;
long b = a;
```

```
int a = 4 + 12b;
```

`+` operation, as the others, requests both operands have same type.

Sometimes it would change temporarily an expression type. In this case you can use type **casting**: simply type of expression you want to convert is followed by new type name enclosed in parenthesis:

```
int a = 4;
long b = (long) a;
```

2.4 Basic operations

Operations are defined with a name, which can be formed only by symbols:

`| $ % & + - * / \ ? ^ < > = | ~ @`

or letters following : (Es. `:new`). All operations have only one (operation symbol precedes always operand) or two (sympol is between them) parameters. These are fundamental operations in Meucci, generated directly by compiler:

Two parameters oerations:

- `+ - * / :mod` Respectively addition, subtraction, multiplication, division and remainder between integer types
- `& | :xor` Bitwise and, or , exclusive or between integer types
- `== < > <= >= !=` Integer comparison

- `&& || ->` Logic operations between logic expressions (returning `boolean`). `&&` returns `true` if and only if both expressions return `true`; `||` returns `true` if and only if at least one expression returns `true`; `->` returns `false` if and only if first expression returns `true` and the second `false`
- `<< >>` Make a left or right shift respectively as bits as return value of second expression. First expression returns integer type, the second `ubyte`
- `+ - * /` These operations work even if both operands return `real`

One parameter operations:

- `++ -- - ~` Increases, decreases by one an integer value, two's complement (number opponent) and ones' complement. Contrary to C/C++, first two operations don't modify expression, so this example

  ```
  ++3;
  ```

  is correct
- `!` It's applicated to logic expressions, returns it negated
- `:sqrt` Applicated to `real` expressions, returns square root

Examples:

```
int a = 4 - 8;
int b = a << u1b;
boolean x = (++ a > b) -> (12 != 12);
```

In spite of other programming languages, Meucci associativity rules are:

1. One parameters operations have always priority over two parameters operations
2. If two or more one parameter operations are applied to the same expression they'll executed from right to left
3. Two parameters operations are always executed from left to right

This last rule usually makes life difficult for skiled (C) programmers. For example this expression:

```
3 + 4 * 6
```

returns 42 instead of 27 as expected. To resolve it you can use parenthesis

```
3 + (4 * 6)
```

First parameter elision

Suppose we have this two code lines:

```
int a = 4;
a = a + 2;
```

At second line expression on the left side of `+` and that on the left side of `=` are equal, so it's possible to remove it obtaining:

```
int a = 4;
a =+ 2;
```

We note that between operation symbol and `=` symbol there isn't any space, or it'll be interpreted as a one parameter operation. It isn't (still) possible to elide parameter of one parameter operation.

2.5 Control statements

2.5.1 if and if-else statements

These istructions are executed sequentially, but we usually need that some of these are executed only if certain condition has been satisfied. Like all other programming languages, Meucci provides `if` statement that execute a logic expression and execute next istruction (or block of istructions) if and only if expression returns `true`.

An example of `if` usage:

```
int x;
...
if(x > 0){
 x =- 1;
}
```

Istruction `x =- 1` will be executed if and only if logic expression `x > 0` is satisfied.

There's also `if-else` statements:

```
int x;
...
if(x > 0){
 x =- 1;
}
else{
 x =+ 1;
}
```

In this case if `x > 0` then it executes `x =- 1`, else it executes `x =+ 1`.

2.5.2 while and for statements

while statements executes several times an istruction (or istructions block) until secified logic expression returns `false`.

```
int x;
...
while(x > 0){
 x =- 1;
}
```

Istruction `x =- 1` is executed until `x > 0` returns `false`, in other words `x` is decremented until it becomes zero, then it comes out of the loop. It notice that `while` tests logic expression before executing relative istruction, so if `x` contains a negative number it's left unchanged.

for statement is very similar to while statement, but it accepts other three istructions as parameters: the first must be expression, declaration or assignment executed before enetering into block, the second a logic expression (like in while statement) whereas the third an expression or assignment (called **step** istruction) executed before control. These parameters are separated by ; . This statement:

```
for(int x = 0; x < 10; x =+ 1){
 println(x);
}
```

is equivalent to

```
int x = 0;
while(x < 10){
 println(x);
 x =+ 1;
}
```

Variables declared inside for statement have limited visibility inside own statement, so cannot be used outside it.

2.5.3 break and continue

Inside iteration statements (while and for) are available these istrucions:

- `break` Leaves immediately loop
- `continue` In while statement jumps execution to logic evauation, whereas in for statement to step istruction.

Chapter 3

Modules

3.1 Module creation

Every operation, function or type has to be contained in some module, and in every file can be one and only one module. To create a module it uses **modulo** keyword, followed by its name and body (containing functions and types) enclosed by curly brackets.

```
modulo Mode{
...
...
}
```

3.1.1 Modules importing

Any module can import functionalities (functions, operations, types, ...) from other modules, simply importing them. In order to import modules it's necessary to have, other than libraries containing codes, its **importation file** (ending with .in or .tin) generated during compilation. These files have to be in a well-known place by compiler.

It uses **depends** keyword to tell compiler which modules import, separated by spaces.

```
modulo Mode depends Mod1 Mod2 Mod3{
...
```

3.2 Extern variables and initializer

Modules need often to store datas they mustn't be destroyed afther functions end or be passed to calling function. **Extern variables** are variables declared inside module but outside any function or type.

```
modulo Mode{
 int vesterna1;
 ushort vest2;
 ...
 ...
}
```

Every function or operation can access to extern variables of its own module, but other modules can't.

You can't perform a declaration-assignment for extern variables because there aren't stored inside any function, so there's the risk of using unclear datas. You can use a special istruction block, called **static** block, that is executed once when module is loaded in memory:

```
modulo Mode{
 int vesterna1;
 ushort vest2;
 static{
  vesterna1 = 0;
  vest2 = u3s;
 }
 ...
 ...
}
```

Chapter 4

Functions and operations

4.1 Funzions and operations declaration

To define functions you hane to write its return type (or **void** if it doesn't return datas), followed by name and its parameters enclosed by parenthesis and separated by commas. Functions parameters are declared as if they are variables. Finally write function body enclosed by curly brackets.

4.1.1 Return istruction

`return` istruction leaves from function and return control to calling procedure. It's followed by expression whose return value is returned to calling procedure, but if function hasn't return value you can use return istruction without any expression. If a function hasn't return value you can omit return istruction **at last line**.

Function declaration examples:

```
ubute A(){
 return u1b;
}

int absoluteValue(int a){
 if(a >= 0)
  return a;
 else
  return - a;
}

void nothing(ushort a, ushort b){
 if(a < u200S)
  return;
```

```
 a =+ b;
}
```

Operations are declared in the same manner of functions, remembering they can only have one or two parameters.

```
long +-@ (long a, long b){
 ...
}
```

Parameters in Meucci are always passed by value.

4.2 Functions overloading

Like C++, Meucci permits functions (and operations) overloading. This is possible because Meucci compiler distinguishes functions not only by them nams, but also by input parameters types. Every time it's used an overloaded function compiler decides which function use analysing expressions passed as parameters return values.

Return type is **not** used for discerning functions, so this example

```
...
int funz(ubyte a)
...
uint funz(ubyte v)
...
```

will generated an error when it'll be compiled.

4.3 Recursion

You can use recursion (technique that allows function to call itself) in Meucci program, simply adding a normal function call inside same function body.

This alghorithm calculates the great common divisor of two unsigned integers using recursion:

```
uint gcd(uint a, uint b){
 if(b == u0)
  return a;
 else
  return gcd(b, a :mod b);
}
```

Chapter 5

Reference types

5.1 Reference type creation

We've seen in previous chapters primitive types, saying they are defined by compiler. **Reference type** contrary to them are sreated by programmers inside modules and can be exported whith them.

Contrary to primitive type variables, when it's declared a reference type variabled compiler doesn't allocated memory for the object, but for a **pointer pointing that object datas**. Futhermore reference objects (datas contained in a reference type variable) ccan ontain inside other variables called **fields**.

Suppose we want to create a Point type, an object representing a classic point on plane. It's characterized by x-coordinate and y-coordinate (supposing both are integer numbers). So our Point type declaration (a type has to be declared before using it) will be:

```
type Point{
 real x;
 real y;
}
```

We've used **type** keyword to declare it, followed by name and by fields each of them must be finished by ; .

To each field can be associated one or more **modifiers**, that are divided in:

- **memorization** modifiers say how field must be stored in memory. Current memorization modifiers are `explicit`, `ghost` and `override`
- **access** modifiers delimit data accesses. Current access modifers are `read` and `shadow`

Every field can have at most one memorization modifier and at most one access modifier, then memorization modifier must precede access modifier if exists.

In next sections it will talk about these modifiers.

5.2 Reference variables

Like primitive types, you can devlare reference type variables in the same manner of primitive type variables:

```
type Fraction{
 int num;
 int den;
}
...
...
Fraction var;
```

To access reference variable fields it uses dot (.) symbol between variable name (or any expression returning an object of that type) and field name you want to access:

```
int i = var.num;
var.den = - 234;
```

5.2.1 Objects allocation

Every time you declare a reference variable, actually you create a variable containing a pointer that should point to object but, because we haven't already created a valid object, it points to random unclear memory location.

To avoid it overwrites critial informations (in exceptional cases the entire system could crash) you should initialize pointer to points your object. If you don't want to create an object you should use **null** constant (you can use it also with `pt` variables:

```
Fraction var = null;
pt pointer = null;
```

But if you want to create a valid object Meucci provides some pseudo-operations you can use in order to create and use objects:

1. **:stack** allocates object in your stack. This allocation method is very fast, but the object will be automatically destroy when program leaves block where it's been created

2. **:new** allocates object in heap. This allocation method is slower than :stack but object won't be destroyed automatically, so this objects can be returned by functions/operations. When you want to destroy it you have to use **:destroy** pseudo-istruction

3. **:static** statically allocates object. It's as fast as :stack, but like :new it won't be destroyed when out of creation block. Object is created **once**,

even if ypu put it into a cycle or a function called more times. You **mustn't** try to destroy it, or program could crash at any time. It's often used in modules initializers.

Examples:

```
Fraction f = :stack Fraction();
Ref g = :new Ref();
Fraction h = :static Fraction();
...
:destroy g;
```

5.2.2 Costructors and destructors

Previous code will generate an error when it's compiled, because :new, :stack and :static only allocate space in memory but don't initialize objects fields, leave them in unclear state. There are particular functions defined in types that inizialize fields, and are called **costructors**. Every object must have at least one costructor (also if it's empty). Inside a costructor you can access object fields with a special variable called **this** of same type of the object you want to initialize. Costructors can have parameters, like functions, and two different costructors of the same object must have different parameters.

A **destructor** is another function defined in object and it's called when you destroy the object. Contrary to costructors you can define one and only one destructor in an object and it can't have any parameter. You can use `this` also in destructors.

Costructors must have **init** as name, while destructors **end**. In both cases you mustn't write return type (it can be considered as if it's `void`) and you end them with ; symbol.

```
type Fraction{
 int num;
 int den;
 init(int num, int de){
  this.num = num;
  this.den = de;
 };
 init(int n){
  this.num = n;
  this.den = 1;
 };
 end(){

 };
}
...
```

```
...
Fraction f = :new Fraction(1, 2);
Fraction g = :stack Fraction(3);
```

NOTE: When an object allocated with `:stack` is destroyed (for example when program exits from a block), destruction will be called automatically.

5.3 Access functions, explicit and ghost modifiers

We consider now Fraction type, defined in Fractions module. If we want to access to a field of a Fraction type variable defined in another module, actually we don't access directly to field but **it's called a particular function** that updates it.

If you don't want to use this functions and to allow other modules to access directly to that field you must use the memorization modifier **explicit**:

```
type Fraction{
 explicit int num;
 explicit int den;
 init(int num, int de){
  this.num = num;
  this.den = de;
 };
 init(int n){
  this.num = n;
  this.den = 1;
 };
 end(){

 };
}
```

Sometimes you need access functions in order to protect datas in object fields. To overwrite access functions it creates a block afther field declaration, then if it wants to overwrite read function it creates a function named **get**, else for write function **set**. Those functions have to satisfy these conditions:

- get function must have field type as return type and mustn't have any input parameter
- set function must have `void` as "return type" and one input parameter of the same type of its field

Both function types can use `this` variable.

For Fraction type: any fraction denominator must be non-zero, but any user can put inside field zero causing logic errors. We use access functions for avoiding this:

```
type Fraction{
 int num;
 int den{
  void set(int val){
   if(val == 0)
    this.den = 1;
   else
    this.den = val;
  }
 };
 init(int num, int de){
  this.num = num;
  this.den = de;
 };
 init(int n){
  this.num = n;
  this.den = 1;
 };
 end(){

 };
}
```

Inside Fractions module accesses to Fraction fields will always be direct. If you want to access with access functions also in same module in which type is defined you shoud use double dot (..) in place of dot (.):

```
var..den = 3;
```

In previous example it's used access functions to access to real fields (allocated in memory). With access functions you can also **simulate** a field, in other words you can create access functions without allocate memory space for that field. You can do this using **ghost** memorization modifier. When you create a ghost field you're forced to create both access functions.

In Fraction type you can create a new ghost field called `floor` that returns integer part (although is righter creating a new function instead of using ghost field). Because it isn't needed to allocate space for it you can use ghost modifier:

```
type Fraction{
 int num;
 int den{
```

```
  int get(){
   return this.den;
  }
  void set(int val){
   if(val == 0)
    this.den = 1;
   else
    this.den = val;
  }
 };
 ghost int floor{
  int get(){
   return num/den;
  }
  void set(int v){

  }
 };
 init(int num, int de){
  this.num = num;
  this.den = de;
 };
 init(int n){
  this.num = n;
  this.den = 1;
 };
 end(){

 };
}
...
...
Fraction var = :new Fraction(5, 2);
int integerPart = var.floor;
```

5.3.1 explicit types

If you want to improve access performance you will declare explicit all fields of that type. But other unused structures (like vtables) will be created reducing performances. Declaring the entire type explicit you avoid to create all useless structures that reduce access speed or increase memory usage

```
type explicit Struct{
```

In explicit types all fields are (implicitly) explicit, and if you use other access modifiers compiler generates an error.

However explicit types can extend or be extended by only explicit types (see inheritance section for more informations) and they **can't have any destructor**.

5.4 Access with parameters

In spite of other programming languages, Meucci allows to access fields with parameters. Consider IntArray type, an array of integer numbers. If you need to access data you need an integer parameter (called in this case index). To secify access parameters of field you should:

1. write after field name parameters types in order, separated by commas and enclosed by square brackets
2. write **both** access functions with this parameters in the same order and before write parameter of `set` function

This is pseudo code of IntArray type:

```
type IntArray{
 int len;
 pt pointer;//Punta alla memoria dell'array
 ghost int elem[int]{
  int get(int index){
   if((index >=0) && (index < len)){
    'returns element'
   }
   else
    return 0;
  }
  void set(int val, int index){
   if((index >=0) && (index < len)){
    'sets element to var'
   }
  }
 };
 init(int len){
  'allocs space'
 };
 end(){
  'frees speca'
 };
}
```

You use square brackets also to access that field:

```
IntArray array = :new IntArray(5);
array.elem[3] = -34;
int v = array.elem[2 + 1]; // v == -34
```

5.5 Access modifiers: read e shadow

Sometimes it wants to avoid or limit accesses to a field because that field may contain critical data. Accesses modifiers manage accesses to it:

- **shadow** prevents other modules from accessing field. All shadow fields are also explicit, so if you put different memorization modifier compiler generates an error

- **read** prevents only from writing it, not reading. If that field isn't explicit it'll generated only read function

So our IntArray type becomes:

```
type IntArray{
 read int len;
 shadow pt pointer;//Points to data
 ghost int elem[int]{
  int get(int index){
   if((index >=0) && (index < len)){
    'ritorna elemento'
   }
   else
    return 0;
  }
  void set(int val, int index){
   if((index >=0) && (index < len)){
    'setta elemento a val'
   }
  }
 };
 init(int len){
  'allocates space'
 };
 end(){
  'frees space'
 };
}
```

5.6 packed fields

Suppose we want to create a type with 127 distinct real types, we've three options. The first is to write manually all 127 real fields changing their names, the second is to create an array with 127 elements, but it'll generate high overhead and many memory problems because Meucci hasn't garbage collector (contary to Java). Meucci provides the third option, that is to use **packed** keyword followed by numbers of fields you want to create:

```
type tti{
 real e packed 127;
}
```

You can access to each field with an **uint** parameter

```
var.e[u59] = 34.89;
```

packed parameters can't have any memorization modifier or access function, you should threat it as if it's explicit. Also it's guaranteed all datas of a packed field are placed in contiguos memory spaces.

5.7 Inheritance

Like object-oriented programming languages, Meucci supports (**single**) inheritance. If you want to extend a type you'll put after supertype name **extends** keyword followed by subtype name:

```
type superT extends subT{
...
}
```

Supertype'll inherits all suptype fields, included access functions. If supertype is placed in a different module from subtype, it can't access shadow fields and could only load read fields.

5.7.1 supercostructors calling

When supertype is initialized there're some fields it couldn't be initialized (like shadow fields). To solve it it's need to call subtype costructor to initialize them with **super** pseudo-function, that must be the first istruction inside costructor:

```
 ...
 init(){
  super(...);
  ...
 }
```

```
...
```

5.7.2 override modifier

Supertype can modify access functions of inhertied fields. To overwrite access functions of a field it's used memorization modifier **override** with the (possible) access modifier, then it writes only new access functions.

Obviously you can't overwrite explicit or shadow fields, and you can only overwrite get function of read fields.

Chapter 6

Template

In previous chapter we've created IntArray type, an array of integer number. If we want to create character array we'll creater another type named CharArray, similarly with ULongArray and so on. In most cases changes only type used while code in the same. Similarly with functions that operate with them.

So in this cases is useful not creating a single type or function, but a **family of types or functions** with the same code, in which each function or type is uniquely determined . A family of types is called **type template**, in the same manner is defined **function template**, while that parameters are called **template parameters**. You can't create operations template.

When you compile modules in which are declared types or functions template is generated a new file ending with .tin, you have to use it in the same manner of .in files.

6.1 Type parameters and number parameters

To create template types or functions, their names must be followed by all template parameters used enclosed by square brackets. There're two types of template parameters

- **type** parameters: it can be name of a generic type. Parameter name follows **typ** keyword. You can add some limitations about types it can represent: with **number** type can be only integer number (long, ushort, int, ...), with **reference** only reference types. With reference limitation you can force it to extend a reference type whose name follows **extends** keyword.
- **number** parameters: it can be any integer **unsigned** constant. Its name follows **num** keyword and precedes a number expressing its type (0 is ubyte, 1 is ushort, 2 uint, 3 ulong). You can add limitations using ¿ and ¡ symbols, followed by constants, that represent upper and lower limits.

Example:

```
type temp[typ T reference extends Fraction, num N 3 > 2 < 4]{
 init(){
 ...
 }
...
}
```

defines for temp two template: T a reference type parameter extending Fraction, N ulong number parameter maior than 2 and minor than 4 (can be only 3).

```
temp[Intero, 3] var = :new temp[Intero, 3]();
//Allocates variable of type temp with T=Intero and N=3
```

6.2 Funzioni template

There are also **functions of template**, or rather functions defined by compiler accepting template parameters and returning a new template parameter:

- **SIZEOF** Accepts a type template and returns dimension of memory allocated for a variable
- **DIMENSION** Accepts a reference type template and returns dimension of that object (allocated with :new, :stack, ...)
- **SUM** Adds two or more number parameters
- **PROD** Multiply two or more number parameters

When you use a function of template, it's to be preceded by `#` symbol and its parameters must be enclosed by parenthesis

```
uint vay = #SUM(u2, 3);
```

Appendix A

Array and StaticArray

Two useful type templates are **Array** (array Java-like) and **StaticArray** (array C-like), both defined in **Arrays** module . Their code is useful for understanding how to use template parameters:

```
modulo Arrays depends Memory{
 type Array[typ T]{
  explicit read uint length;
  shadow pointer[T] memory;
  ghost T elem[uint]{
   T get(uint index){
    if(index < this.length){
     pointer[T] val=somma[T](this.memory, index);
     return val.el;
    }
    else{
     return (T) null; //Error
    }
   }
   void set(T va, uint index){
    if(index < this.length){
     pointer[T] val=somma[T](this.memory, index);
     val.el=va;
    }
   }
  };
  init(uint size){
   pt memory=allocate(#SIZEOF(T)*size);
   this.memory=(pointer[T])memory;
   this.length=size;
  };
```

```
  end(){
   free((pt)this.memory);
  };
 }
 type explicit StaticArray[typ T, num L 2]{
  T elem packed L;
  init(){

  };
 }
}
```

Example of use:

```
Array[ubyte] b = :stack Array[ubyte](u2 + u1);
StaticArray[ubyte, 3] sb = :stack[ubyte, 3]();
b.elem[u1] = u12b;
sb.elem[u0] = b.elem[u1] +u1B;
```

Difference is that Array dimension can be determined only at run-time, memory must be allocated dinamically, while StaticArray dinemsion is known at compile-time (with the number template parameter L) so memory is allocated statically by compiler.

```
type explicit pointer[typ TT]{
 TT el;
}
```

pointer[T] type is simply a pointer pointing to T object (T* in C).