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 Funzioni di accesso e modificatori explicit e ghost

Prendiamo sempre in esempio il tipo Frazione, e definiamolo all'interno di un modulo di nome Frazioni. Se ora volessimo accedere ad un campo di una variabile di tipo Frazione dichiarata in un altro modulo, in realta non accediamo direttamente al campo in questione, ma **viene chiamata una particolare funzione** che per default accede in lettura o in scrittura al relativo campo.

Se per motivi di performance si vuole evitare la creazione delle funzioni di accesso e permettere anche a moduli esterni di accedere direttamente al campo si utilizza il modificatore di memorizzazione **explicit**:

```
type Frazione{
 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(){

 };
}
```

Ci sono situazioni in cui pua far comodo utilizzare le funzioni di accesso per proteggere i dati memorizzati nei relativi campi. Per sovrascrivere le funzioni di accesso di default basta creare un blocco subito dopo il campo e quindi, se si vuole sovrascrivere la funzione di lettura, dichiarare una funzione di nome **get** mentre, per sovrascrivere la funzione di scrittura, dichiarare una funzione di tipo **set**. Queste funzioni devono pera soddisfare le seguenti caratteristiche:

- La funzione get deve avere come tipo di ritorno il tipo del campo e non pua avere parametri di input

- La funzione set deve avere void come "tipo di ritorno", e deve avere un solo parametro di input dello stesso tipo del campo

Entrambe le funzioni possono utilizzare this per accedere all'oggetto.

Ritorniamo al nostro tipo Frazione: una frazione deve avere necessariamente il denominatore diverso da 0 per esistere, ma cosa come escritta potrebbe essere posto tranquillamente a 0 con il rischio di generare errori logici. Utilizziamo le funzioni di accesso per prevenire a questa eventualita:

```
type Frazione{
 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(){

 };
}
```

Da notare che all'interno del modulo Frazioni ogni accesso ai campi di variabili Frazione sara sempre diretto. Se si vuole che un accesso sia mediato dalla relativa funzione si deve utilizzare al posto del punto singolo (.) due punti affiancati (..)

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

Le funzioni di accesso sono molto potenti: negli esempi precedenti quando si dichiara un campo viene allocato lo spazio sufficiente per contenere il dato. In Meucci epossibile dichiarare campi **senza allocare spazio al campo stesso**, ma regolando gli accesso tramite le funzioni di accesso. Si crea quindi una sorta di campo fantasma, in cui viene data l'illusione dell'esistenza del campo quando gli accessi vengono tutti evasi dalle funzioni di accesso. Ovviamente per un campo ghost si dovranno specificare entrambe le funzioni di accesso.

Supponiamo di aggiungere un campo fantasma al nostro tipo Frazione che ritorna la parte intera del nostro oggetto (benché sarebbe pia corretto definire una funzione distinta che calcoli la parte intera). Si utilizzera il modificatore di memorizzazione **ghost** per creare un campo fantasma:

```
type Frazione{
 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 parteIntera{
  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(){

 };
}
```

Se un campo eghost, allora anche all'interno del modulo l'accesso sara mediato dalle funzioni.

5.3.1 Tipi explicit

Per migliorare le performance di accesso di un oggetto o per poter calcolare esattamente le dimensioni dello stesso si sarebbe tentati di dichiarare explicit

ogni campo del tipo. In realta vengono generate comunque altre strutture inutili (basta pensare alle vtable per gestire l'ereditarieta). Per impedire la creazione di queste altre strutture si pua rendere l'intero tipo explicit ponendo tale parola chiave tra il nome e la parole type:

```
type explicit Struct{
```

In un tipo explicit l'accesso a tutti i campi sara sempre diretto, e l'unico modificatore di memorizzazione ammesso eexplicit (che viene naturalmente ignorato).

C'epera un prezzo da pagare: Un tipo explicit pua estendere solo tipi explicit e per estendere un tipo explicit anche il sovrattipo dovra essere explicit. Inoltre **non si potra specificare un distruttore**. Questi argomenti verranno trattati nel capitolo dell'ereditarieta.

5.4 Accesso con parametri

A differenza di altri linguaggi di programmazione, Meucci permette di accedere ai campi anche attraverso dei parametri. Per comprendere meglio consideriamo il tipo IntArray, un array di interi. Per accedere ad un generico dato memorizzato nell'array si utilizza un parametro intero, che in questo specifico caso viene chiamato indice. Tramite le funzioni di accesso epossibile specificare parametri aggiuntivi per accedere al campo, che in Meucci deve possedere il modificatore **ghost**. Per specificare i parametri di accesso bisogna:

1. Scrivere subito dopo il campo in ordine i tipi dei parametri di accesso racchiusi tra parentesi quadre e separati da una virgola
2. Specificare entrambe le funzioni di accesso con i relativi parametri di accesso in ordine e sempre dopo l'eventuale parametro di scrittura per la funzione set.

Ecco lo pseudo-codice del nostro tipo IntArray:

```
type IntArray{
 int len;
 pt pointer;//Punta alla memoria dell'array
 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){
  'alloca spazio'
 };
 end(){
  'libera spazio'
 };
}
```

Per accedere al campo si devono specificare i parametri all'interno di parentesi quadre:

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

5.5 Modificatori di accesso: read e shadow

Sempre per protezione, spesso si vuole impedire o limitare l'accesso ai campi da parte di moduli esterni. Per limitare l'accesso si utilizzano i seguenti modificatori di accesso:

- **shadow** Impedisce completamente l'accesso al campo ai moduli esterni. Specificando un campo shadow implicitamente lo si rende explicit, quindi specificare un campo sia shadow che con un modificatore di memorizzazione diverso da explicit genererà un errore del compilatore
- **read** Impedisce solo la scrittura al campo, non la lettura. Se il campo non eexplicit, verra generata solo la funzione di accesso in lettura, generando un errore nel caso si voglia creare anche una funzione di accesso in scrittura

Cosa il nostro tipo IntArray diverra:

```
type IntArray{
 read int len;
 shadow pt pointer;//Punta alla memoria dell'array
 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){
   'alloca spazio'
  };
  end(){
   'libera spazio'
  };
 }
```

5.6 Campi packed

Supponiamo di voler creare un tipo con 127 campi distinti di tipo real, abbiamo tre opzioni. La prima edi scrivere manualmente tutti i 127 campi cambiando il nome di ciascuno di essi, la seconda edi creare un array di 127 elementi, ma cia creerebbe un overhead non trascurabile e una gestione della memoria difficoltosa, in quanto in Meucci non epresente il garbage collector come in Java. La terza opzione edi utilizzare la parola chiave **packed** seguita da 127, cosa il compilatore generera automaticamente tutti i campi

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

L'accesso a ciascun campo viene effettuata come se avesse un parametro di tipo **uint**

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

Se si dichiara un campo packed, non sara possibile assegnargli un modificatore di memorizzazione né inserire funzioni di accesso, il campo dovra essere trattato come se fosse explicit.

Come vedremo nel prossimo capitolo, la lunghezza di un campo packed pua essere un template di tipo numerico, rendendo questo strumento molto utile in varie circostanze.

5.7 Ereditarieta

Come nei linguaggi orientati ad oggetti, il linguaggio Meucci supporta il meccanismo dell'ereditarieta (**singola**). Per estendere un tipo basta inserire, dopo il nome del sovrattipo, la parola chiave **extends** seguito dal nome del sottotipo:

```
type sovraT extends sottoT{
...
}
```

Il sovrattipo ereditera tutti i dati del sottotipo, comprese le funzioni di accesso. Se il sovrattipo ecreato in un modulo diverso da quello in cui estato dichiarato il sottotipo, il modulo non potra accedere ai membri shadow ereditati e potra accedere solo in lettura ai campi read ereditati. In compenso, l'accesso ai dati ereditati sara diretto, a meno che il campo non sia ghost o vengano usati i due punti.

5.7.1 Chiamata dei sovracostruttori

Quando si deve inizializzare un sovrattipo enecessario inizializzare anche i campi del sottotipo a cui non si avrebbe accesso (come i campi shadow). Per ovviare a cia la prima istruzione di un costruttore derivato deve essere una chiamata ad un costruttore del sottotipo, utilizzando la pseudo-istruzione **super**:

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

5.7.2 Il modificatore override

La sovracclasse pua modificare le funzioni di accesso dei campi ereditati sostituendole con altre pia adatte. Per sovrascrivere le funzioni di accesso si riscrive nel sovrattipo il relativo campo con modificatore di memorizzazione **override** e con l'eventuale modificatore di accesso e dei parametri di accesso, e quindi si scrivono le nuove funzioni di accesso che andranno a sostituire quelle vecchie.

Non epossibile effettuare l'override di campi explicit o shadow, e per i campi read epossibile modificare solo la funzione di lettura.

Chapter 6

Template

Nel capitolo precedente abbiamo creato un tipo IntArray, un array di interi. Se ora volessimo creare un array di caratteri si dovrebbe creare un tipo CharArray, analogamente con ULongArray e cosa via. Si nota che la maggior parte del codice elo stesso, cambia al massimo il tipo immagazzinato. Discorso analogo vale per le funzioni che operano su tali tipi.

Per questo ecomodo creare non un solo tipo o funzione, ma una **famiglia di tipi e funzioni** con codice simile, in cui ciascun tipo o funzione eunivocamente determinato da uno o pia parametri. Una famiglia di tipi viene detta **tipo template**, analogamente si dice **funzione template**, mentre questi parametri di accesso sono detti **parametri template**. Non si possono creare allo stato attuale operazioni template.

Quandi si compila, questi tipi e moduli non vengomo memorizzati all'interno di un file .in, ma il compilatore generera dei file .tin, generalmente pia pesanti, che devono essere utilizzati come i normali file di importazione.

6.1 Parametri tipo e parametri numerici

Per creare un tipo o una funzione template bisogna far seguire al nome del tipo/funzione, tra parentesi quadre, tutti i parametri template utilizzati. Esistono due tipi di parametri template:

- Parametri **tipo**: il parametro pua essere il nome di un tipo qualunque. Bisogna far precedere al nome del parametro la parola **typ**. Si possono aggiungere altre limitazioni ai valori ammissibili, poste dopo il nome del parametro: con **number** il tipo pua essere solo un tipo intero (long, ushort, int, ...), mentre con **reference** pua essere solo un tipo reference. Se si imposta la limitazione reference, si pua obbligare il parametro ad assumere solo nomi di tipi che estendono un determinato tiporeference tramite la parola **extends** seguito dal sottotipo

- Parametri **numero**: il parametro pua essere una qualunque costante **senza**

segno. Bisogna far precedere al nome la parola chiave **num** e far seguire un numero che ne esprime il tipo (0 eubyte, 1 eushort, 2 euint, 3 eulong). Si possono aggiungere limitazioni ai valori che il parametro pua assumere tramite i simboli ¿ e ¡ seguiti da costanti, che esprimono l'estremo inferiore e superiore del parametro.

Esempio:

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

definisce per il tipo temp due parametri template: T parametro tipo reference che estende il tipo Frazione, N parametro numerico di tipo ulong maggiore di 2 e minore di 4 (pua essere solo 3).

L'esempio seguente mostra come dichiarare e allonare memoria ad una variabile del tipo ottenuto dal tipo template temp con T = Intero (che estende il tipo Frazione semplicemente ponendo den=1) e N = 3:

```
temp[Intero, 3] var = :new temp[Intero, 3]();
//Le costanti utilizzate come parametri numerici sono
//sempre unsigned, anche se manca il prefisso u
```

6.2 Funzioni template

Ci sono inoltre delle **funzioni template**, ovvero funzioni definite dal compilatore che prendono come input parametri template e ritornano un parametro template:

- **SIZEOF** Accetta come parametro un parametro tipo e ritorna come parametro numerico la dimensione in byte della quantita di memoria allocata durante la dichiarazione
- **DIMENSION** Accetta un parametro tipo reference e ritorna come parametro numerico la dimensione in byte dell'oggetto
- **SUM** Accetta due o pia costanti o parametri numerici e ritorna un parametro numerico che ela somma degli input
- **PROD** Accetta due o pia costanti o parametri numerici e ritorna un parametro numerico che eil prodotto degli input

Quando viene utilizzato una funzione template, deve essere preceduta dal simbolo cancelletto `#` e i parametri devono essere racchiusi tra parentesi tonde e separati da virgole

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

Appendix A

Array e StaticArray

Due tipi template molto utili e utilizzati sono **Array** (un array in Java) e **StaticArray** (un array in C), definiti entrambi nel modulo **Arrays** incluso di default in tutti i progetti. Il loro codice emolto utile per comprendere il funzionamento dei parametri template, per questo verra riportato integralmente in questa appendice:

```
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; //Condizione di errore
    }
   }
   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(){
   //Non e' necessario il costruttore,
   //in quanto l'allocazione della memoria
   //e' statica
  };
 }
}
```

```
Array[ubyte] b = :stack Array[ubyte](u2 + u1);
StaticArray[ubyte, 3] sb = :stack[ubyte, 3]();
b.elem[1] = u12b;
sb.elem[0] = b.elem[1] +u1B;
```

La differenza sta nel fatto che la dimensione del tipo Array si pua determinare solo in fase di esecuzione, e quindi enecessario allocare dinamicamente la memoria, mentre la dimensione di StaticArray edeterminata gia in fase di compilazione (tramite il parametro template numerico L) e quindi il compilatore conosce gia la dimensione dell'array.

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

Il tipo template pointer[T] va inteso come un puntatore a elementi (non per forza oggetti) di tipo T.

Appendix B

Funzionalita future

Ecco un breve elenco delle funzionalita che forse verranno aggiunte nelle prossime versioni del compilatore

- Gestione eccezioni
- Possibilita di definire nuovi tipi primitivi
- Libreria multithreading
- Protocolli, ovvero astrazioni che permette il caricamento dinamico di moduli
- Possibilita all'interno di funzioni di accesso di chiamare le precedenti funzioni di accesso, al fine di migliorare la protezione dei dati membro