Let’s consider the following classes and their members

**class** A

f1

f3

**end**

**class** B **extends** A

f2

**override** f3

**end**

Particular signatures of f1, f2, f3, and all other members are skipped as they are out of the scope at the moment

When we compile classes A and B we have the following relation between classes and members

Table I.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **f1$A** | **f2$B** | **f3$A** |
| **A** | A.f1 |  | A.f3 |
| **B** | A.f1 | B.f2 | B.f3 |

Notation **member$class** defines seed and origin, where seed is the version of the member (actual declaration) in the class (origin) which contains this declaration. So, **f1$A** means that member f1 was first time declared in class A and it can be overridden (redefined) in derived classes (descendants). We may consider pair seed and origin as the top of the member versions – set of first declaration and all overriding ones. So, pair **seed$origin** can be treated as unique hash code of the top of the member hierarchy. So, every filled cell of the matrix refers to the particular member (method or attribute) to be called presented as the name and class which contains declaration of the member

Typically if we consider this matrix by rows and limit members to member-functions (methods) only we have VMT – virtual method table which is attached to every class descriptor at runtime. But if we are start considering the same matrix as set of columns then per column we have MST – member selection table and this is translated into the offset in this table to get address of method or offset for class attribute to be called or accessed. Let’s try to explore MST approach a bit deeper and see how it works. Effectively we have just transposed the matrix

Table II.

|  |  |  |
| --- | --- | --- |
|  | **A** | **B** |
| **f1$A** | A.f1 | A.f1 |
| **f2$B** |  | B.f2 |
| **f3$A** | A.f3 | B.f3 |

Step1. Separate compilation of A and B. As A does not depend on other classes and its compilation is straightforward while B relies on A and results of separate compilation of any class should store its dependencies (it can be hash codes of class interfaces, full signatures or time stamps – these are implementation details which are not considered at the moment). So, we build VMT-like table per class where all member declared in this class are referenced like below

Table III-IV

|  |  |  |
| --- | --- | --- |
| A own members | **f1$A** | **f3$A** |
| **A** | A.f1 | A.f3 |

|  |  |  |
| --- | --- | --- |
| B own members | **f1$A** | **f2$B** |
| **B** | A.f3 | B.f2 |

So, these vectors are just subsets of Table I, having removed inherited members

Step2. So, when we build executable or dynamic library which uses both A and B together we produce a set of another vectors (MST) and they allow supporting dynamic binding at runtime in the similar way to what VMT does. These are just rows of Table II

a: A = **new** B

a.f1 /\* Dynamic type of ‘a’ is B and from the vector called f1$A we need to get the effective address for the class B \*/

We skip all possible optimizations how to effectively implement such call as we are mostly interested in step 3

Step3. During program execution we load class C which was separately compiled and has its own row oriented vector. When C was compiled it was verified that it was correct within the particular interface of it base classes (parents)

**class** C **extends** B

**override** f2

f4

**end**

Table V

|  |  |  |
| --- | --- | --- |
| C own members | **f2$B** | **f4$C** |
| **C** | C.f2 | C.f4 |

So, if version of B which is already loaded is Ok for C then we need to integrate C into the existing system. Effectively we need to update Table II with the information from Table V and build Table VI as a result

Table VI

|  |  |  |  |
| --- | --- | --- | --- |
|  | **A** | **B** | **C** |
| **f1$A** | A.f1 | A.f1 | A.f1 |
| **f2$B** |  | B.f2 | C.f2 |
| **f3$A** | A.f3 | B.f3 | B.f3 |
| **f4$C** |  |  | C.f4 |

So, theoretically we may load classes dynamically and integrate them into the working environment which supports dynamic binding.

Few more general observations analyzing the content of the Table VI (MST vectors)

1. Each column answer the question which members can be accessed when some data entity (variable or constant) has static type equal to the column type

a: A; a.f1 and a.f3 are valid

b: B; b.f1, b.2, and b.f3 are valid

c: C; c.f1, c.f2. c.f3, and c.f4 are valid

1. Each row clear highlights if we need dynamic dispatch of we know which member version is to be accessed for any dynamic type – so, static dispatch can be applied

f1$A – regardless of the dynamic type of the call target A.f1 is to be called – static dispatch

f2$B – access B.f2 for this = B and access C.f2 if this = C

f3$A – if this = A then access A.f3 otherwise access D.f3

f4$C – this vector has only one element filled – so, static call to C.f4 is the right code

1. The number of columns can be reduced if all abstract classes are removed, classes for which there is no new expression met in the program code (class has no dynamic type). Dead-code elimination allows to remove members which are not accessed in the program code – reducing the number of filled cells in this table (if the member is not accessed just remove all references to it from the MST)

So, such table is good for sematic analysis and for optimizations as well

Class vs interface:

There are numerous explanations why interfaces are different from classes and why programming language needs to have both and distinguish between them two. TBC!

**Formal definitions:**

Type is “In [computer science](https://en.wikipedia.org/wiki/Computer_science) and [computer programming](https://en.wikipedia.org/wiki/Computer_programming), a **data type** (or simply **type**) is a set of possible values and a set of allowed operations on it” (<https://en.wikipedia.org/wiki/Data_type#Definition> )

Type is {DataValues, OperationOnDataValues}

Example:

{{1,2,3}, {+,-,\*,/}} data values are 3 integer numbers and set of operations with names explicitly stated

Integer is {{MinInteger..MaxInteger}, {{+,-,\*,/}}} here we name the type and define its values as a range plus names of operations

In both cases we still have a lot of items which are not fully defined. We have not defined fully operations – we just named them. And we have defined what is MinInteger and MaxInteger as well as what is 1, 2, and 3. Also we have not defined what is range.

T is {{new T(arguments1), new T(arguments2)}, {T.routine1, T.routine2}}

One more attempt to declare type T – let’s try to put all possible variants of how we crate instances (objects) of type T and provide a full list of routines (procedures and functions) which compromise set of operations over objects of type T. Of course we may drop T and new inside curly brackets

T is {{(arguments1), (arguments2)}, {routine1<signature1>, routine2<signature2>}}

So, what we have got – list of all possible calls to constructors (initialization procedures) of type T plus all routines with their signatures. What we cannot list fully is all combinations of constructor calls. So, we need another mechanism of specifying which objects will be valid for the type T taking into account that we do not like to list them all as a general mechanism. And we have to look deeply at what object (instance of some type) is. If we take one element of the possible type data values and add set of allowed operation we have the object. So, set of all objects is the type of these objects. They all have the same operations while different values.

{1, {{+,-,\*,/}}} - that is object 1

{2, {{+,-,\*,/}}} - that is object 2

Can we use the value of the object as it unique identifier – yes we can. So, 1 is both the name and value of the object. If we have a complex number then the scheme stays the same. 5+6i can be written as 5i6 or (5,6) but 5 as real part and 6 as imaginary one form the value and name of such object.

So, intuitively we see that every object has some internal structure – list of fields or attributes which contain next level values. And these values have types as well. So, if to drill down and reach the end of this hierarchy we have only 2 atomic objects bar and circle, | and O, or 1 and 0. All other objects are constructed from these constituents. What is the type objects 1 and 0 belong to? It was named as bit. So, heer is the first essential definition

Bit is {{0, 1}, {set of operations over bits}}

Can we express it in a form of constructors?

Bit is {{zero(), one()}, {set of operations over bits}}

Why not to name constructor? And we did it.

But let’s make a step back to fields or attributes of an object. The most straightforward way is just to give a name to each attribute

MyObject1 is {{f1 is value11, f2 is value12}, {set\_of\_operations}}

MyObject2 is {{f1 is value21, f2 is value22}, {set\_of\_operations}}

And so on

So, type of all these objects is a matrix. That is another representation of type – matrix of all type instances

Next logical step is to state that column (value11, value21, value31, etc) is a type in fact as we have just dropped set of operations associated with every field value. So, from matrix of objects we come to the vector with types

MyType is {{f1 : Type1, f2: Type2}, {set\_of\_operations}}

That fully explains the notation which is widely used for structures, records, classes, interfaces and so on. Some particular type kinds with have empty set of values and we are used to call them interfaces, some will have operation with no real, effective body and then we start calling such classes as abstract ones again fixing the fact that there could be no real object of this type. But the key thing is that all such types have a complete description of type attributes and routines. Does this entity deserve a special name? Well the name which suits this is class. The wage thing is that for years term class was used together with association that all objects of this type either reference objects (Java and many other languages) or value object (C++). To break this contradicting association the term unit is proposed to denote a type which has a explicit description of its attributes, operation and construction procedures. To deal with terminology ambiguity other terminology is introduced to keep the mess running ☺

Are there type kinds which do not have a explicit description of its attributes and operations (routines) ? Yes they are. Algebraic types are the first example.

Union is T1 + T2 + T3 // union of types

Intersection is T1 \* T2 \* T3 // intersection of types

One more group of types is tuple types

Tuple is (T1, T2, T3)

Another one is anchored ones

Anchor1 is like this

Anchor2 is like someAttribute

Functional (routine) types, called lambda one

Function is (Par1Type, Par2Type):ReturnType

Procedure is (ParType)

They also define a set of values and set of operations but they do not list them explicitly. And another distinction between union (class) types and all others that union types explicitly state their inheritance relation (extend-implements one in Java terms) to other union types

So, it is worth to define what inheritance is. Short and 100% correct definition it is relation between union types. But this does not reflect the nature of this relation. And the nature can be expressed that if B inherits A then B will have some attributes and routines from B, some adopted and some new introduced immediately in B. So, for convenience we will be using the C++ terminology calling unit attributes and routines as members. Member attribute and member routine

Well is it possible that derived (child) unit will have just a subset of base (parent) unit members. Why not? What we cannot do in this case we cannot use object of derived type instead of the object of the base one. Effectively we cannot assign to the object of the derived unit to the base one. So, in other word derived unit does not conform to the base one. It just have a different form or objects. This brings us to the concept of conformant inheritance and non-conformant one. But in any case inheritance means a kind of movement of members of base classes to the derived one

Another important concept which was already mentioned is reference and value entities. When we say object we always mean set of values and set of operations-routines. So, we may notice that we treat object as a value data entity. So, what is a data entity or simply an entity? Local (declared within some routine) variable or constant, unit variable or constant attribute, this, function result are examples of entities. All these entities can be in the form of reference to an object or an object itself (value). And they all have names associated with these entities. Objects can have no name or treat set of its values as its name! Entity is part of notation used by programmer which creating the source code. Objects exit while program is being executed and can be stored not only in RAM but on external storage too. Entities are abstractions expressed in the source code.

When we deal with entities we have only 2 operations – entity creation and ‘dot-call’ operation

a is A or a is new A or a is new A() syntax form is just a sugar

a := expr0

a.foo (expr1, expr2)

b := struct.field

struct.field := expr3

Can we interpret assignment as a form of the ‘dot-call’ – yes!

a.:= (expr0)

b.:=(struct.field)

struct.field.:=(expr3)

So, entity has 2 states – created and not-created-yet. There are cases when we need to declare an entity but not to create it immediately. Keep the name but top have it not attached to a real object. Be non-initialized. (That is null-safety one may exclaim – let it be so, but conceptually different)

So, based on the declaration of an entity in the program source code entities could be attached to an object and detachable (may be either attached to an object or not). How object can be attached to an entity. It can be attached by the runtime of the program like ‘this’ is automatically attached to the currently active object. Or entity declaration may have explicit (a is new A) or implicit (a: A) initialization which attaches this entity to some object. Or it can be a direct assignment of freshly created object or already initialized entity to the entity of interest (a := new A; a := b)