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

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

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** | f1$A | f3$A |

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

So, these vectors are just subsets of Table I

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** | f2$C | f4$C |

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

Table VI

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

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