RUNTIME DATA STRUCTURES FOR OEXPL

Virtual Function Table

Virtual Function Table is a run time data structure that is used to resolve at run time the call addresses of methods in a class. Such mechanism (or other table schemes) needs to be implemented by compilers for languages that support inheritance, subtype Polymorphism and method over-riding. A run time virtual function table is maintained in the memory for each class. The compiler generates code to initialize the table such that each function that can be invoked from the class has an entry in the table. For each function defined or inherited by a class, the table will contain the call address of the function in the code region of memory. Thus, if a function is inherited by a class from its parent and is not over-ridden by the class, the call address stored (will be same in the virtual function tables of both the classes) will be that of the function defined in the parent class.

The OExpL compiler maintains a virtual function table of eight words for each class. Hence, a class can have at most eight member functions. The i-th entry in the table holds the address of the i-th function of the class (identified by the field Funcposition in the class table - see Memberfunclist). While generating code, if there is a call to a function within a class, the compiler simply translates the call to "CALL [address]", where address is obtained from the corresponding virtual function table.

A simple way to place the virtual function table is to use the initial part of the stack region (starting at memory address 4096), ahead of global variables. This is suggested because the compilation of classes will be finished before compiling the global declarations. In this scheme, the virtual function table of the class defined first in the program will be stored in the region 4096-4103, the second in 4104-4111 and so on.

When the compiler encounters the declaration of a variable of a class, it allocates two words of storage for the variable (unlike other variables, where only one word of storage is allocated). The first word is used to store the address of the memory area allocated in the heap for the class (eight words). Space for member fields is allocated in the heap exactly as done with user defined types. We will call this word the member field pointer. The second word is used to store the address of virtual function table. We will call this word the virtual function table pointer.

The key point to note here is that the virtual function pointer of a variable need not always point to the virtual function table of the class of the variable. This is because a variable of a parent class can hold a reference to an object of any of its descendant classes. In such cases, the virtual function pointer will hold the address of the descendant class.

Hence, an assignment of a variable (of one class) to another variable (of the same class or an ancestor class) results in transfer of the contents of both the pointers of one variable to the corresponding pointers of the other. The *new* function sets the virtual function table pointer to the address of the virtual function table of the class specified as the argument to the new function. A little thought would reveal that this implementation will yield the correct execution semantics.

A detailed illustration follows.

Illustration

```
class
A
{
  decl
    int f0();
    int f1();
  enddecl
  int f0() {                   /*Newly defined method*/
      begin
       write("In class A f0");
       return 1;
      end
  }
  int f1() {                   /*Newly defined method*/
      begin
```

```
        write("In class A f1");
        return 1;
      end
  }
}                       /*End of Class Definition A*/
B extends A
{
  decl
    int f0();
    int f2();
  enddecl
int f0() {                    /*f0 of class A is overridden by this meth
     begin
       write("In class B f0");
       return 1;
     end
}
int f2() {                    /*Newly defined method*/
     begin
       write("In class B f2");
       return 1;
     end
}
/* Class B inherits f1 from Class A */

}                       /*End of Class Definition B*/
C extends B
{
  decl
    int f0();
    int f2();
    int f4();
  enddecl
int f0() {                /*f0 of Class B is overridden by this method*/
     begin
       write("In class C f0");
       return 1;
     end
}
int f2() {                /*f2 of Class B is overridden by this method *
    begin
       write("In class C f2");
       return 1;
    end
}
int f4() {                /*Newly defined method*/
    begin
```

```
        write("In class C f4");
        return 1;
     end
}

/*Class C inherits f1 from Class A */

}                         /*End of Class Definition C*/
endclass
```

example.txt hosted with ❤ by **GitHub** **view raw**

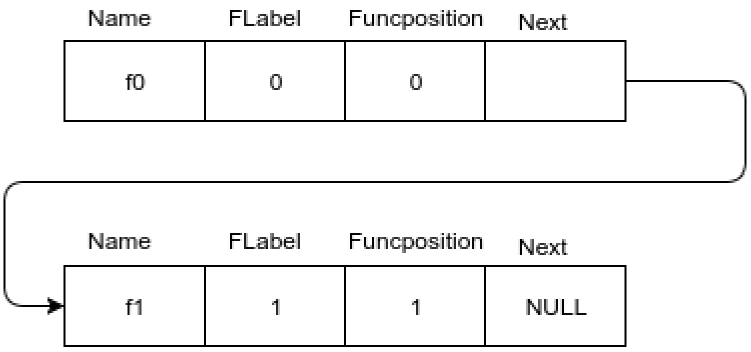
1. Storage for virtual function tables start at 4096 in the stack. Eight consecutive words are allocated for storing the virtual function table of each class. Each word of virtual function table stores the call address of the corresponding function in the class. Thus, a class can contain atmost eight methods. Virtual function tables of various classes are stored in the order in which the classes are declared in the program. As noted earlier, we will allocate the space for global declarations after virtual function tables are allocated.

   By using the *Class_index* field of the class_table entry, the starting address of the virtual function table for that particular class can be computed using the formula 4096 + (Class_index * 8). The class defined first will have *Class_index* zero, the next will have one and so on.

2. When the declaration of each method is found (in a class) a new label is generated for the function and the label is stored in the class table entry for the function at compile time. The compiler also must generate code to store these labels into the virtual function table entry of the function in the corresponding class. (Strictly speaking, an integer value called the "pseudo-address" for the function is stored in the *flabel* field of the *Memberfunclist* entry of the function. When code is generated, for functions with *flabel* values 0,1,2,3, etc., the actual labels placed could be F0,F1,F2,F3 etc to make them more human readable.) The index of the function in the class table (funcposition) and the virtual function table is maintained to be the same.

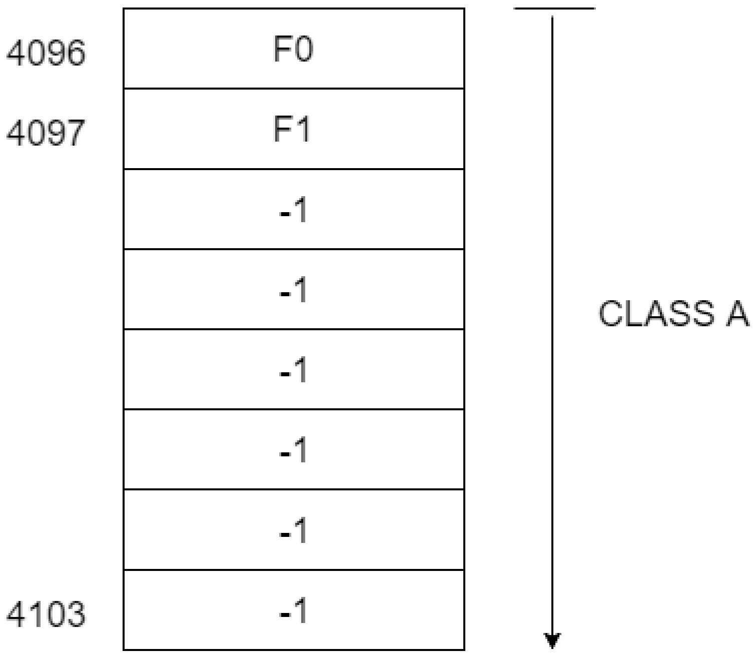
   [Note: It is sufficient to place labels, and not addresses in the virtual function table, as the label translation phase will take care of translating labels to addresses].

In the example given, the function f0 in class A has funcposition 0 and say flabel F0 (we identify flabel 0 with F0) and the function f1 in class A gets a funcposition 1 and say flabel F1 (we identify flabel 1 with F1). The member function list of class A looks as shown in the below figure :



Function Name	Label Name
f0	F0
f1	F1

The virtual function table of class A looks as shown in the figure below. It is constructed using member function list of class A which is shown in the figure above.

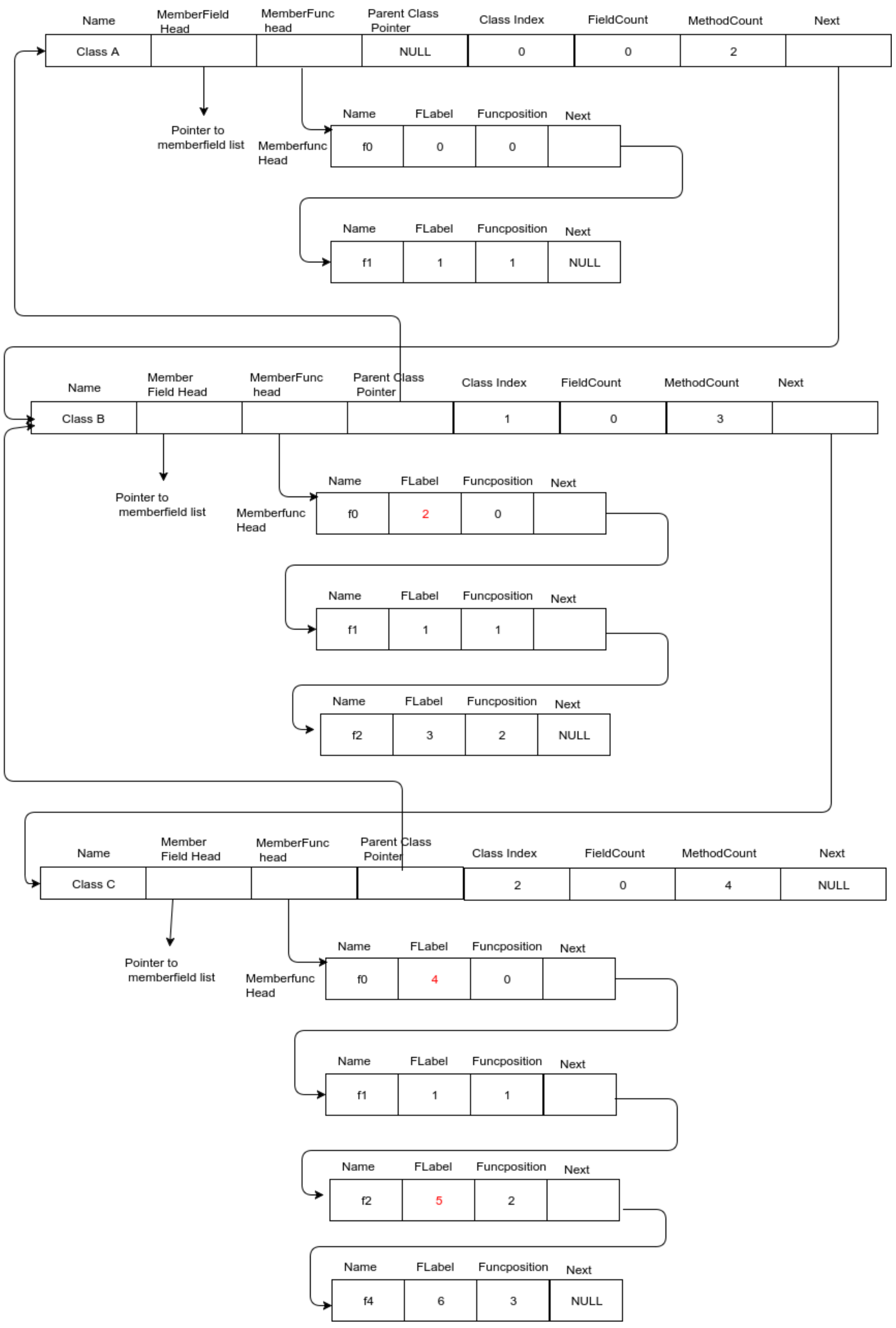


As mentioned earlier, all the labels of the functions will be replaced with addresses during label translation phase.

3. Class B extends class A and over-rides f0(). Further, class B contains the newly defined method f2(). When a class extends another, all the member fields and methods of the parent class are inherited by the derived class, unless over-ridden by a new definition. Since the method f0() is over-ridden by B, a new label will have to be allocated for the function f0() in class B. In the present example, we set the new label to F2. Accordingly, the compiler must update *Memberfunclist* entry of the method f0() in class B with the new *flabel* value. Correspondingly, in the virtual function table entry for class B, the entry for method f0() must be F2 (over-riding F0). The entry for method f1() (label F1) will be inherited from class A. A new label (label F3 in the example) must be generated for the function f2() defined in class B. The labels for each method in class B is shown in the table below for easy reference.

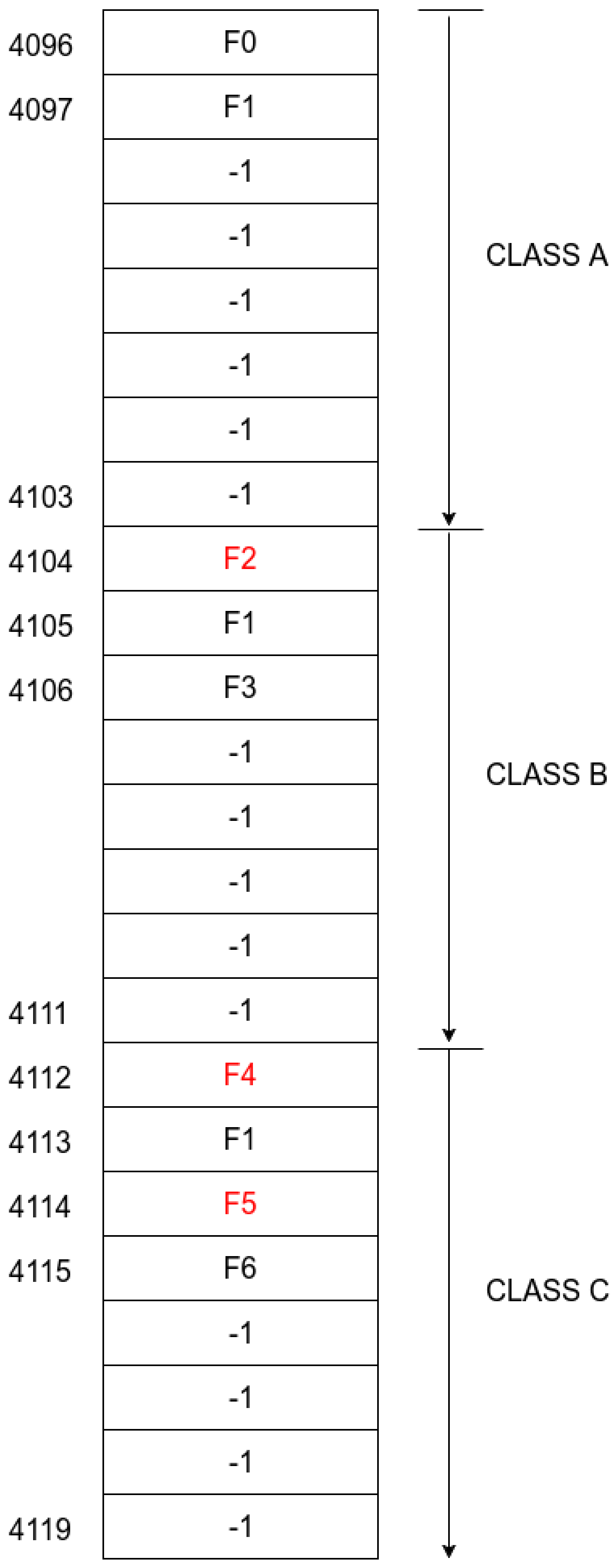
Function Name	Label Name
f0	F2
f1	F1
f2	F3

From an implementation point of view, it will be easier to (generate code to) copy all the virtual function table entries of A to the virtual function table of B and then (generate code to) modify the labels of over-ridden functions/add labels for new functions defined in B. The compilation for class C may proceed similarly. Note that OExpL specification stipulates that the signatures of the over-ridden methods must match exactly with the signature of the original definition in the parent class.



★ Overridden labels are marked in red

The corresponding virtual function tables of all the classes are shown in the figure given below :



★ Overridden labels are marked in red

OEXPL Run Time Binding Tutorial

Github

Contributed By : J.Ritesh
J.Phani
Koushik
M.Jaya
Prakash

Home | About