CINT : Reducing the size of the CINT dictionaries

ROOT TEAM

By Leandro Franco & Diego Marcos

Introduction: What's a dictionary?

3 different ways of C/C++ code execution

Typing in the CINT Prompt

```
root [0] Int_t myInt
root [1] myInt++
(Int_t) (0)
root [2] myInt
(Int_t) (1)
```

Introduction: What's a dictionary?

Loading / Executing an external Script

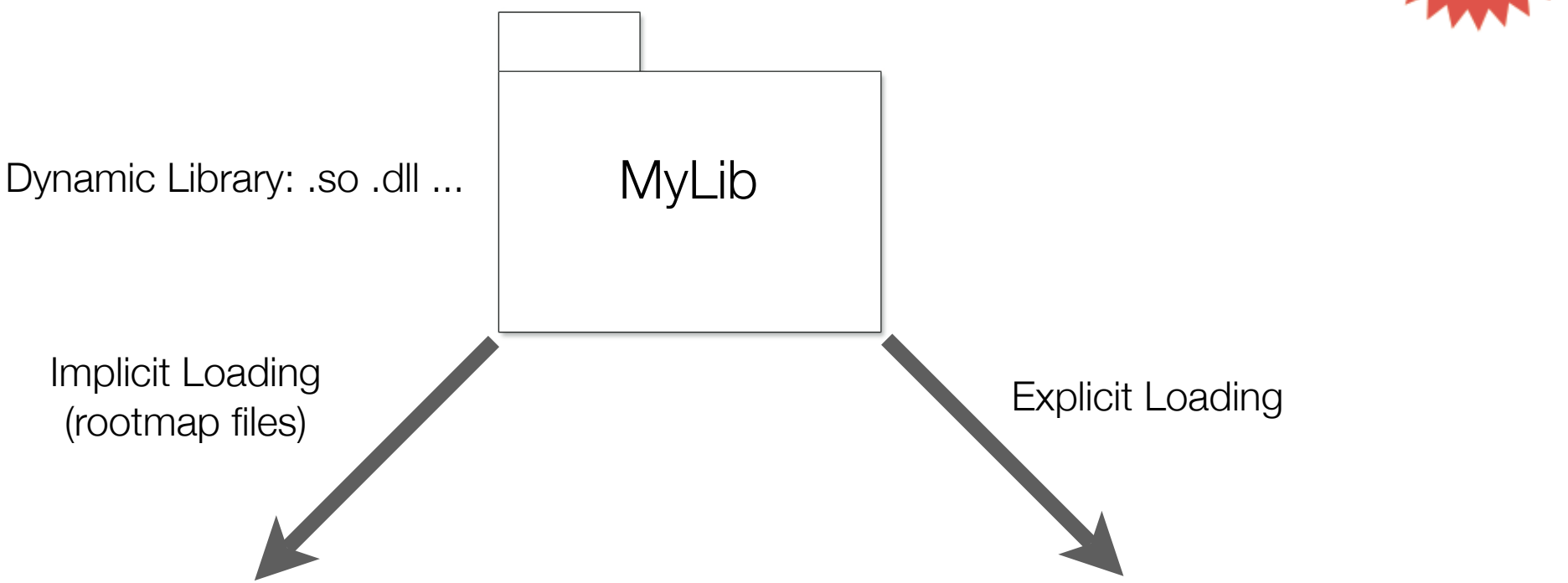
script.C

```
#include <iostream.h>

class MyClass {

public:

   MyClass(){}
  virtual ~MyClass() {}

  void MyClass::PrintMe() const{
     cout << "I'm your class" << endl;
  };

};
```

```
root [0] .L script.C
root [1] MyClass obj
root [2] obj.PrintMe()
I'm your class
```

Introduction: What's a dictionary?

Loading / Executing Compiled Code

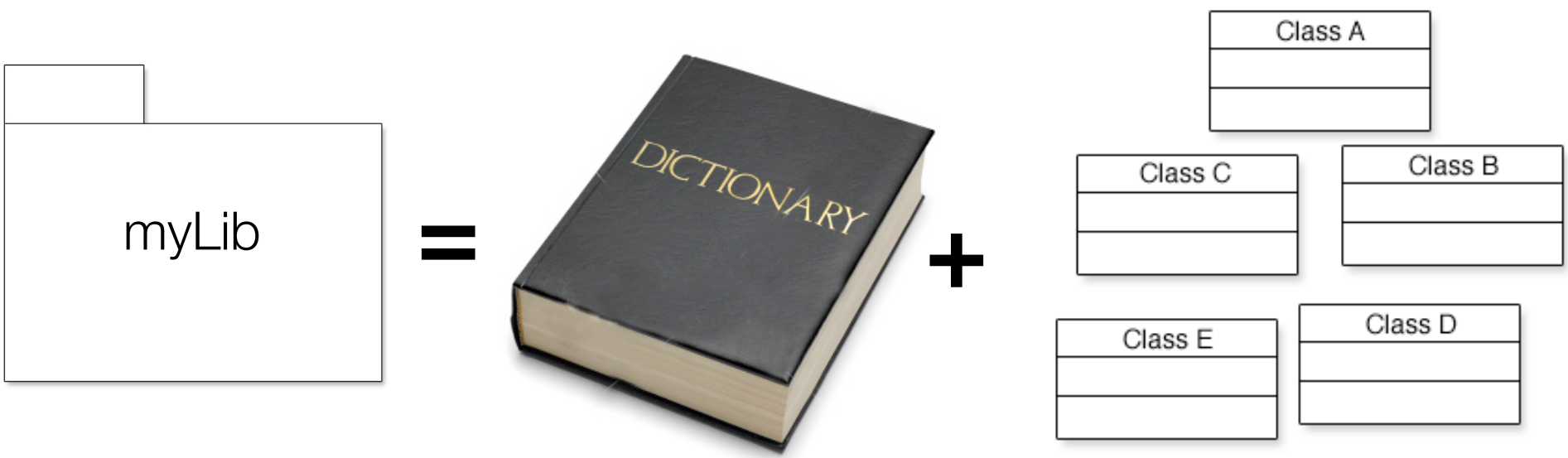



```
root [0] TH1 myHist
```

```
root [0] gSystem->Load("myLib")
root [1] MyClass obj
```

Introduction: What's a dictionary?

But ... Cint needs to know about the classes in the library



Library **Dictionary** Set of classes

Introduction: What's a dictionary?

What is a dictionary made of?
At present, it is made of code, automatically generated for each library.

A dictionary represents about **50%** of the library size

ROOT libraries set is about **75MB** in the default configuration and **100MB** with full support.

46 MB of the **75MB** are due to the dictionaries

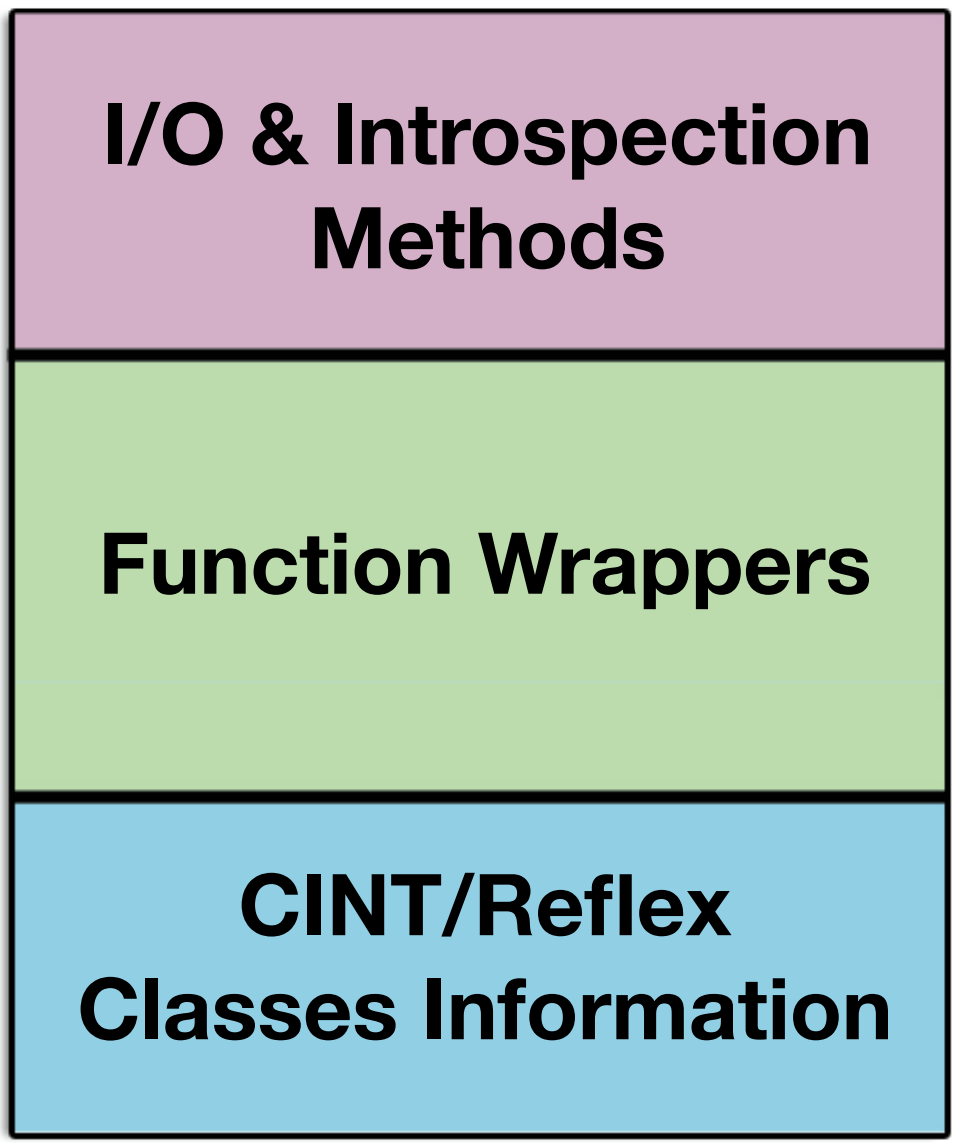
It's definitely a good idea to reduce their size. Isn't It?

Dictionary Dissection

30 %

40 %

30 %



Methods automatically added to the classes

aka Stubs

Information about classes. Members, Hierarchy...

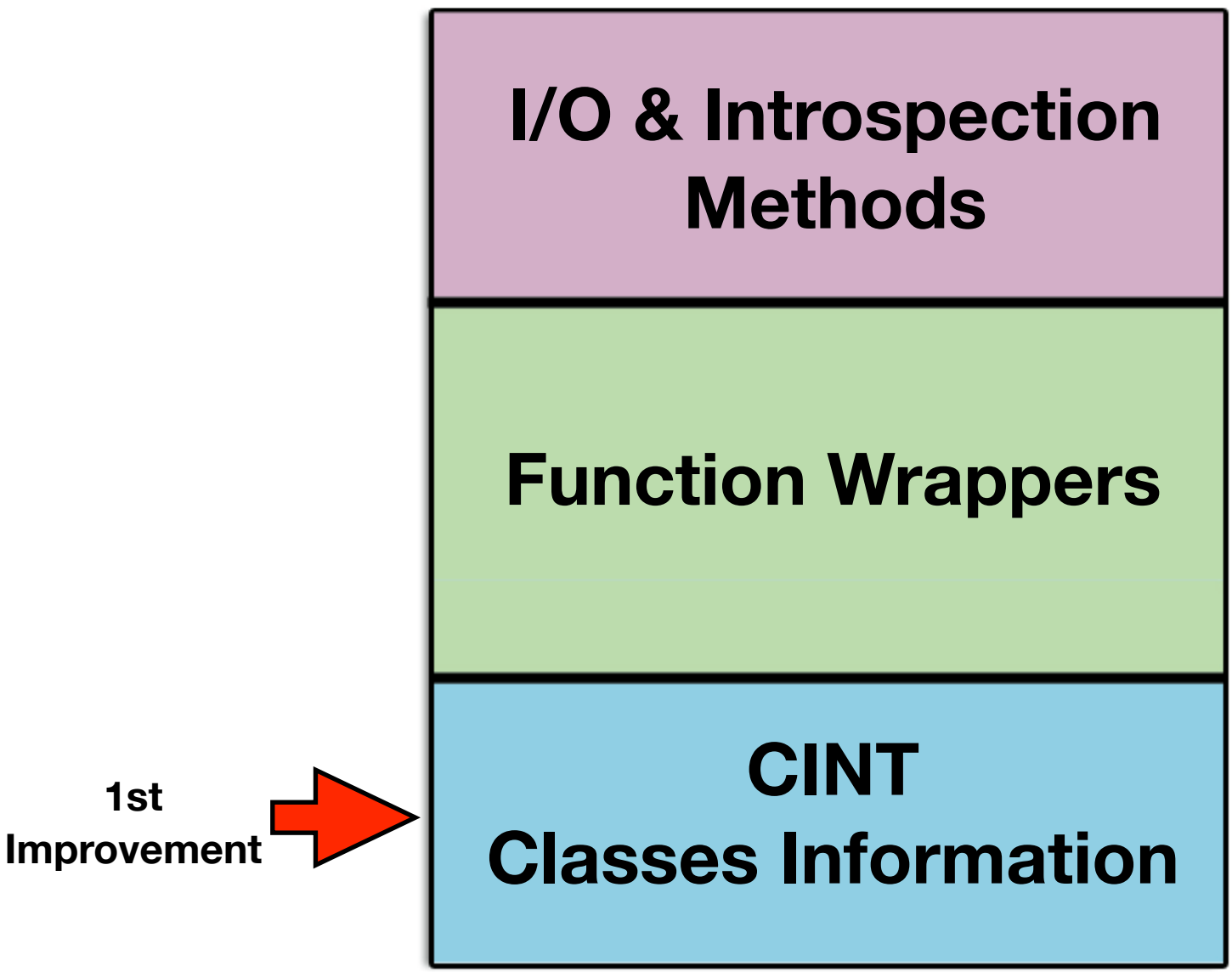
Buyer Be Aware

- The following improvements were realized at a very low level and are compiler and machine dependant. For the moment, it works with GCC and Intel Machines but we expect to extend the support to Visual C++ and other platform very soon.

- Obviously, there's full backward support. These features will be optional at the beginning and is up to the user to test them (they won't be included by default in the dictionary generation).

- For ROOT, it will be the default behaviour when the right conditions are found (right compiler, platform, etc).

1st Improvement



1st Improvement

Data and Function Member information will be stored in **ROOT files**

ROOT files are **more compact** and allow a **smaller level of granularity**

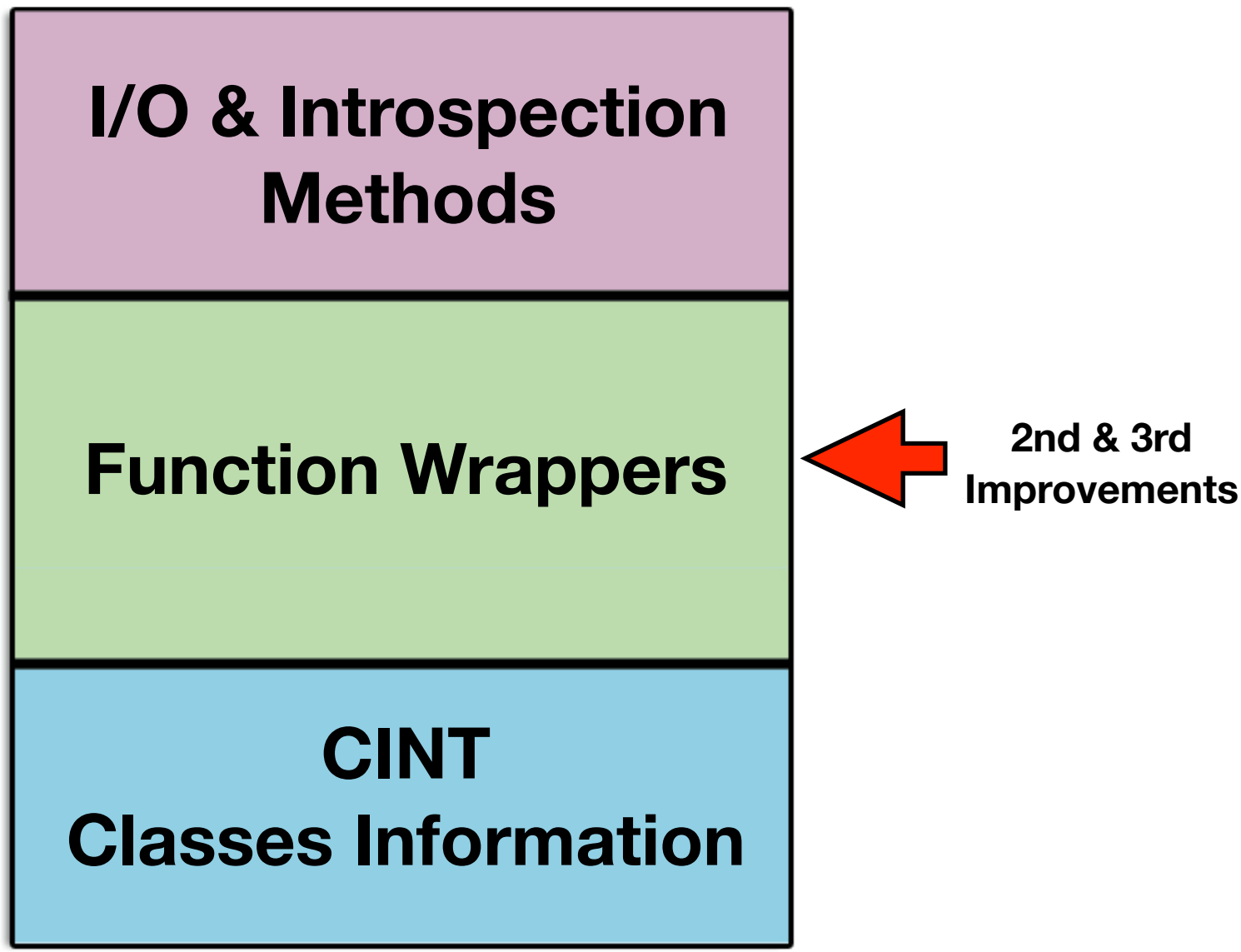
Example - LibHist

Dictionaries With Members Information = **1.8 MB**

Dictionaries Without Members Information = 1.5 MB + 100Kb ROOT File = **1.6 MB** (Only the first step)

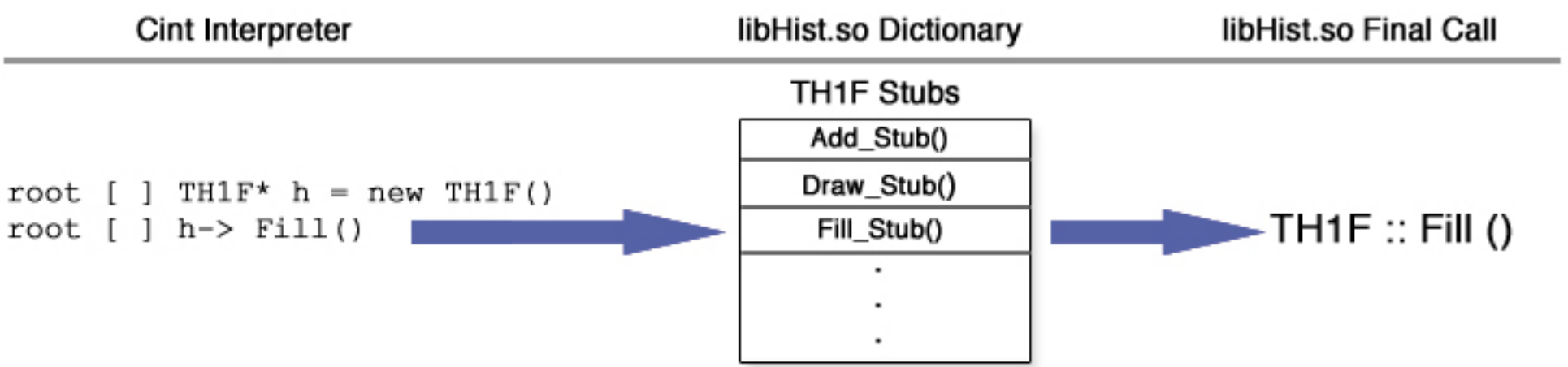
Status: Already Working. We are debugging

2nd Improvement

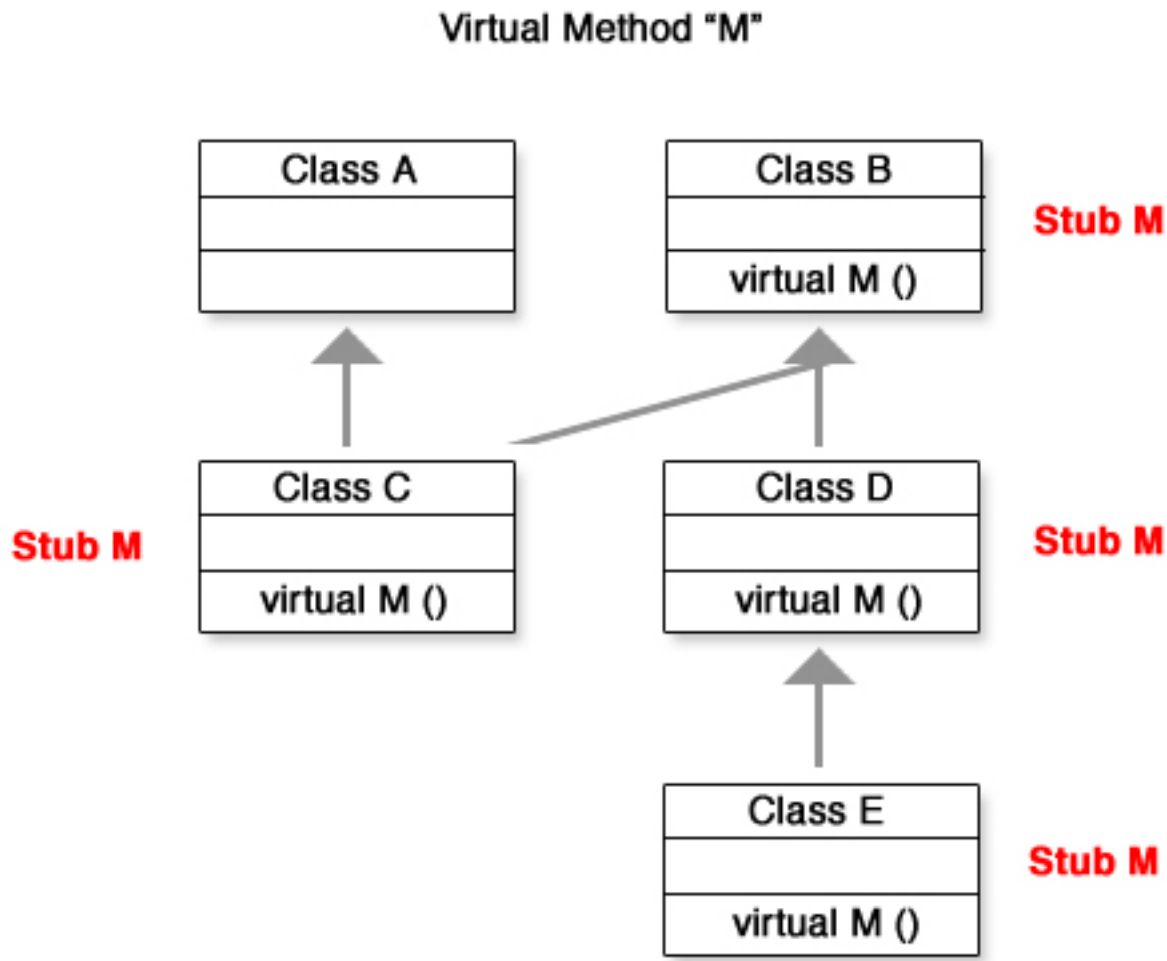


2nd Improvement - Stub Functions

Stubs allow us to call compiled functions in a way independent of the machine and compiler

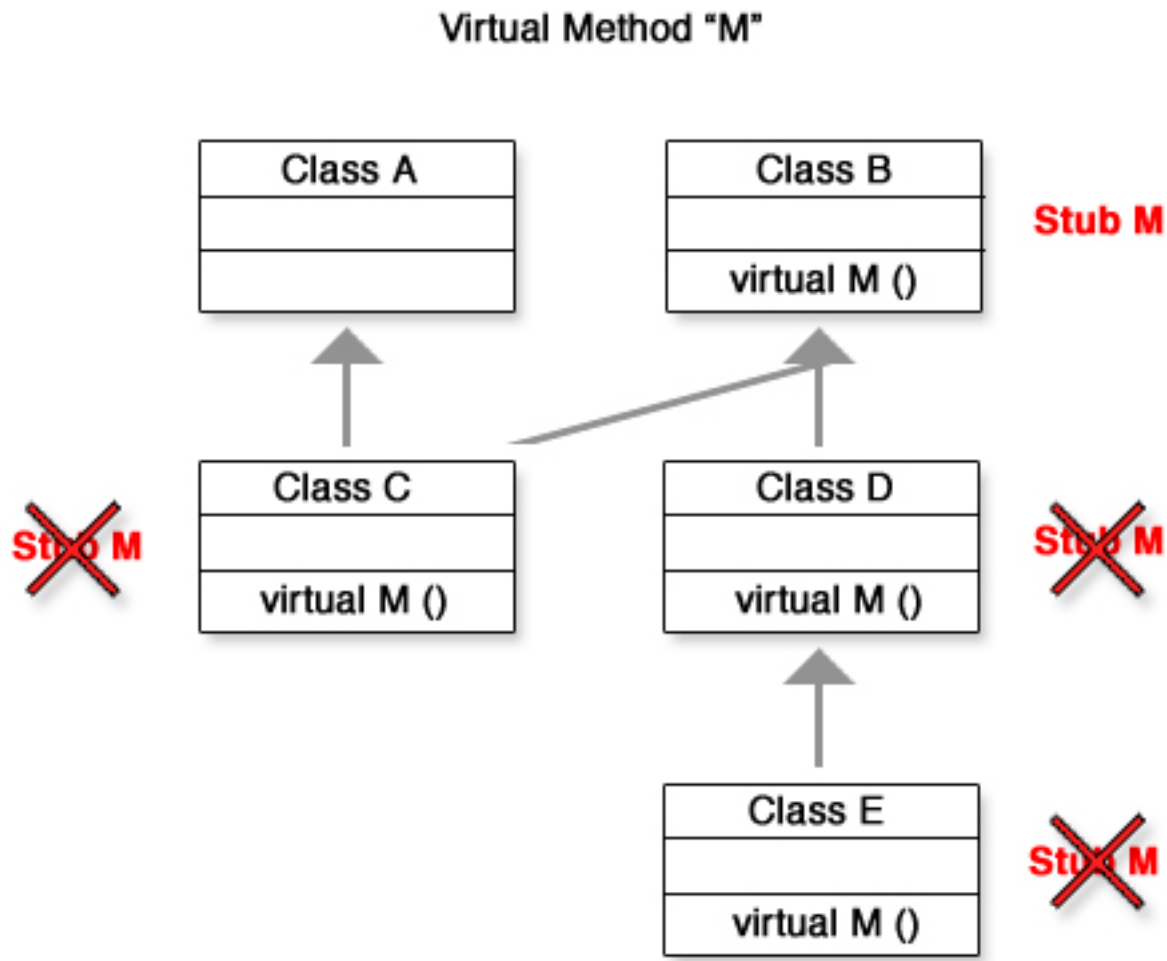


Improvement Scenario and Idea



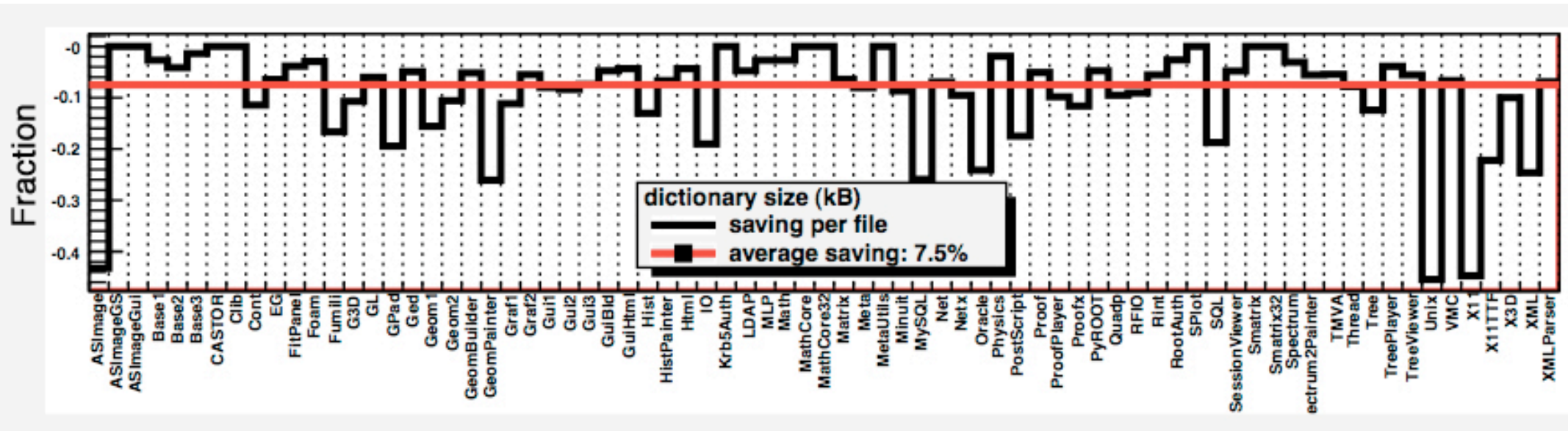
- We only keep the stub in the base class
- Let C++ Virtual function mechanism call the proper method
- This improvement is already in the current ROOT version

Improvement Scenario and Idea



- We only keep the stub in the base class
- Let C++ Virtual function mechanism call the proper method
- This improvement is already in the current ROOT version

Results



We reduce dictionaries by 7.5 %

Those changes are machine independent

Status: it's already in the ROOT version 5.16

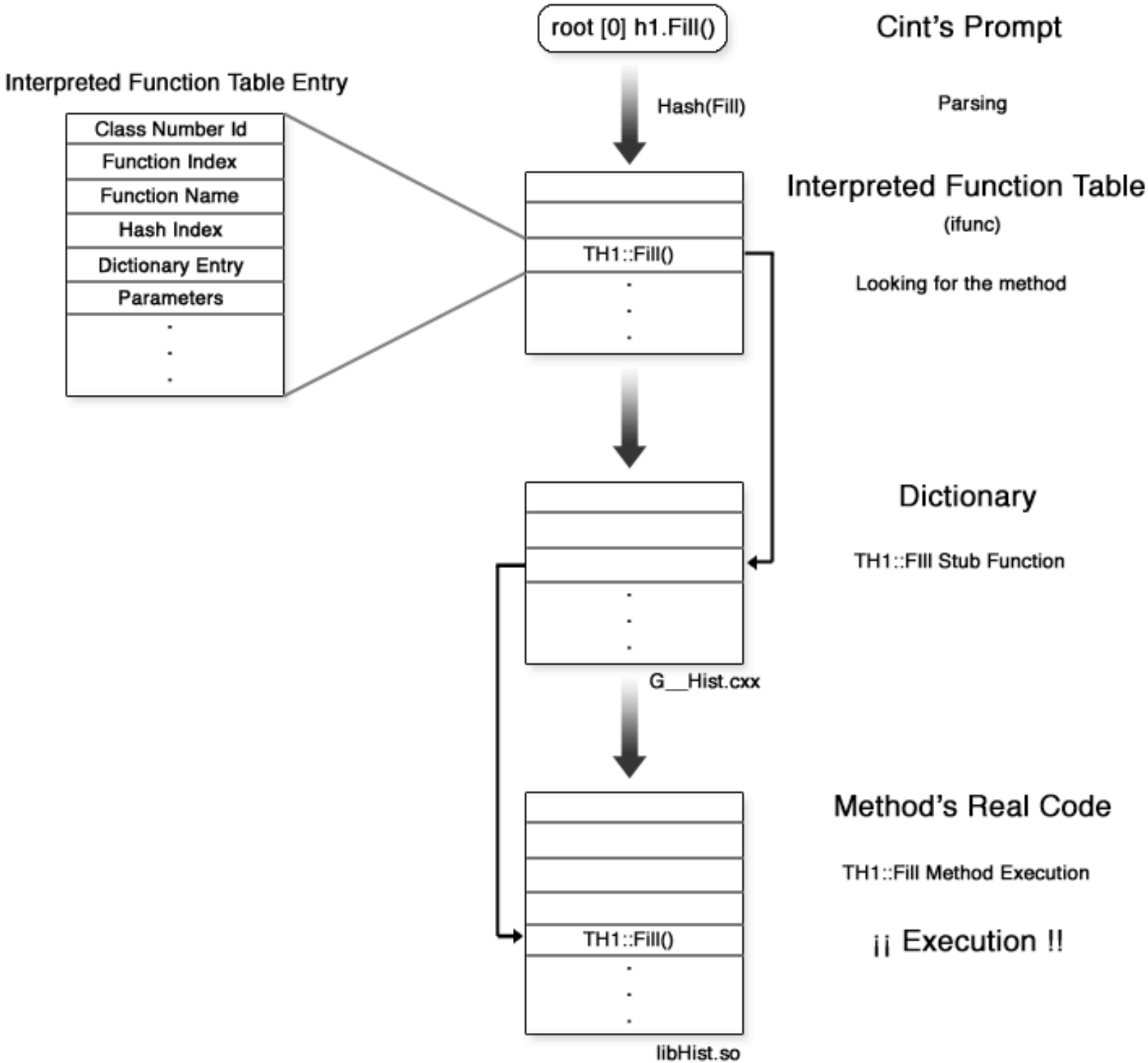
3rd Improvement - Stubs Removal

- Initializing CInt data structures.
  - At loading time we have one “G__memfunc_setup” call for each stub function (in a file like G__Hist.cxx).
  - This function creates a new field in the G__ifunc_table with attributes like name, hash, type, parameters, etc. (Including the pointer to the stub function!)

```
G__memfunc_setup2("Fill",391,G__G__Hist_118_0_39, 105, -1, G__defined_typename("Int_t"),
                  0, 1, 1, 1, 0, "d - 'Double_t' 0 - x", (char*)NULL, (void*) NULL, 1);
```

```
static int G__G__Hist_118_0_39(G__value* result7, G__CONST char* funcname, struct G__param* libp, int hash)
{
   G__letint(result7, 105, (long) ((TH1*) G__getstructoffset())->Fill((Double_t) G__double(libp->para[0])));
   return(1 || funcname || hash || result7 || libp);
}
```

We can see both functions in the dictionary... for every single method!!!


root [0] h1.Fill()
Cint's Prompt
Interpreted Function Table Entry
Class Number Id
Function Index
Function Name
Hash Index
Dictionary Entry
Parameters
Hash(Fill)
Parsing
TH1::Fill()
Interpreted Function Table
(ifunc)
Looking for the method
Dictionary
TH1::FIll Stub Function
G__Hist.cxx
Method's Real Code
TH1::Fill Method Execution
TH1::Fill()
¡¡ Execution !!
libHist.so

What is a function call in CInt

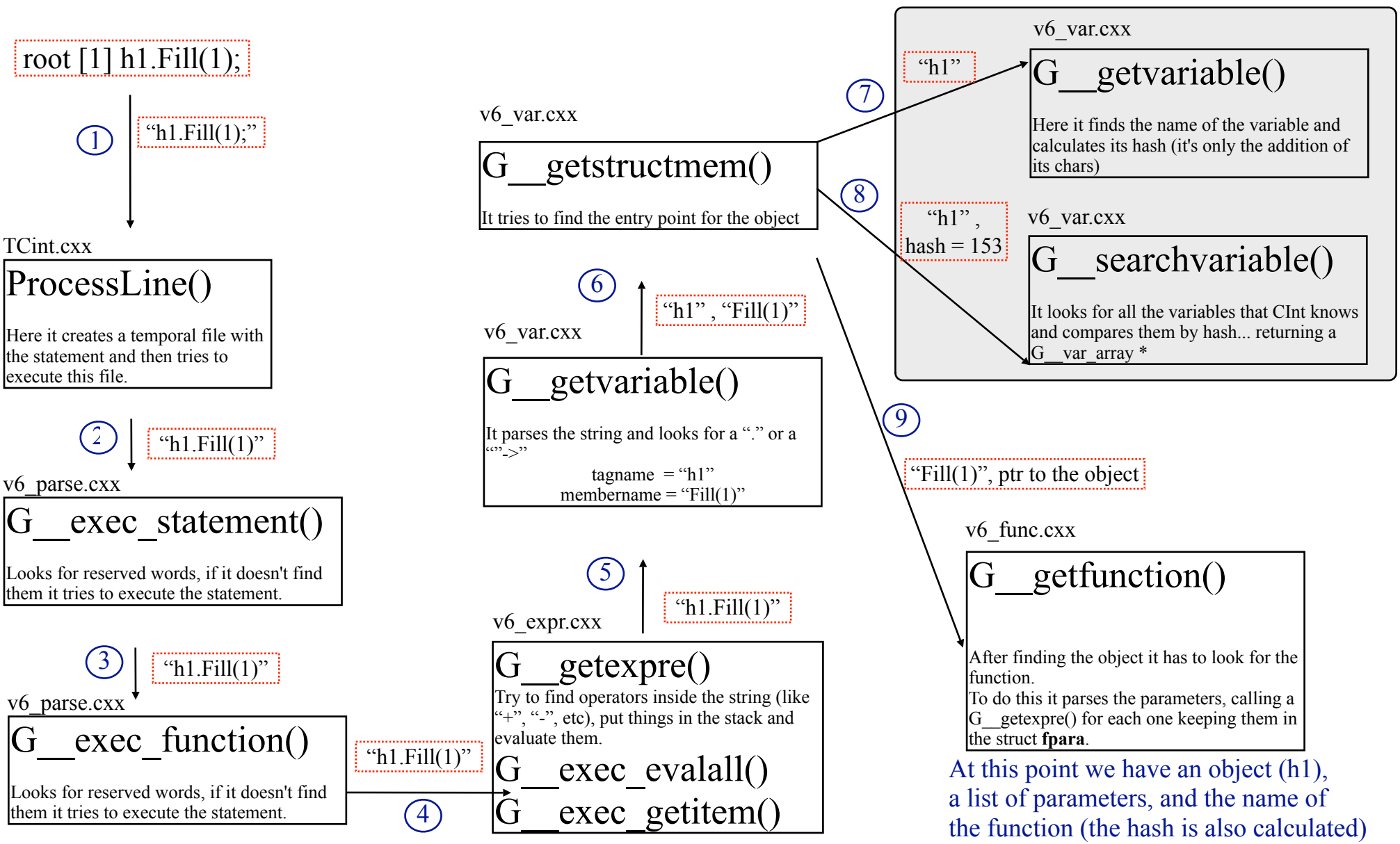
A CInt call is something similar to a C/C++ statement that is usually executed from ROOT

```
2.05b$ root.exe
 ****************************************
 *                                      *
 *       W E L C O M E  to  R O O T     *
 *                                      *
 *   Version   5.15/03  14 February 2007*
 *                                      *
 *  You are welcome to visit our Web site*
 *         http://root.cern.ch          *
 *                                      *
 ****************************************

FreeType Engine v2.1.9 used to render TrueType fonts.
Compiled on 21 February 2007 for linux with thread
support.

CINT/ROOT C/C++ Interpreter version 5.16.18, February
9, 2007
Type ? for help. Commands must be C++ statements.
Enclose multiple statements between { }.
root [0] TH1F h1("h1", "h1", 10, 10, 10)
root [1] h1.Fill(1);
```

A simple example of function call


```
root [1] h1.Fill(1);
```

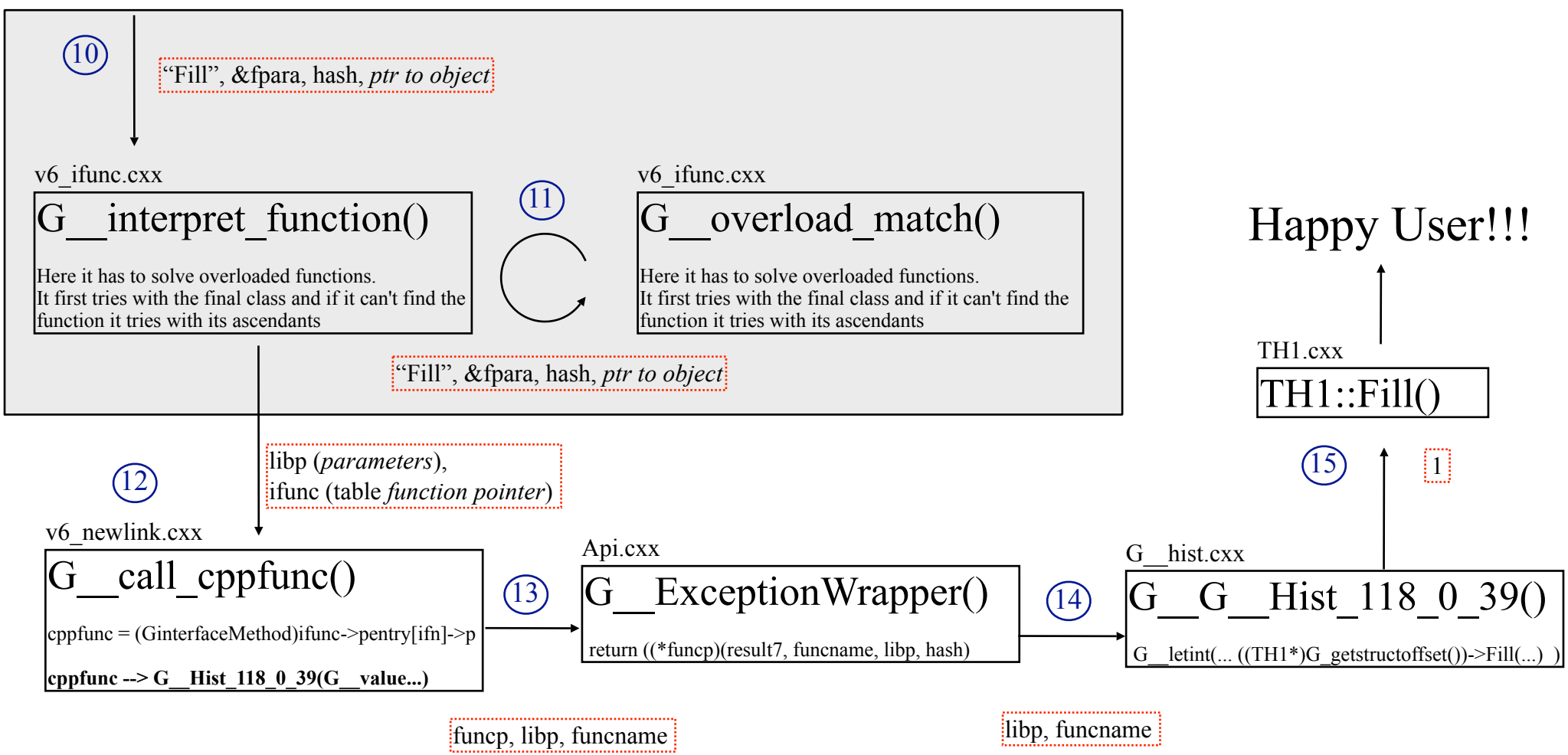
Now... the long story



At this point we have an object (h1), a list of parameters, and the name of the function (the hash is also calculated)

Almost there...

At this point we have an object (h1),
a list of parameters, and the name of
the function (the hash is also calculated)



Replacing stubs with dynamic calls

```
G__memfunc_setup2("Fill",391,G__G__Hist_118_0_39, 105, -1, G__defined_typename("Int_t"),
                  0, 1, 1, 1, 0, "d - 'Double_t' 0 - x", (char*)NULL, (void*) NULL, 1);
```

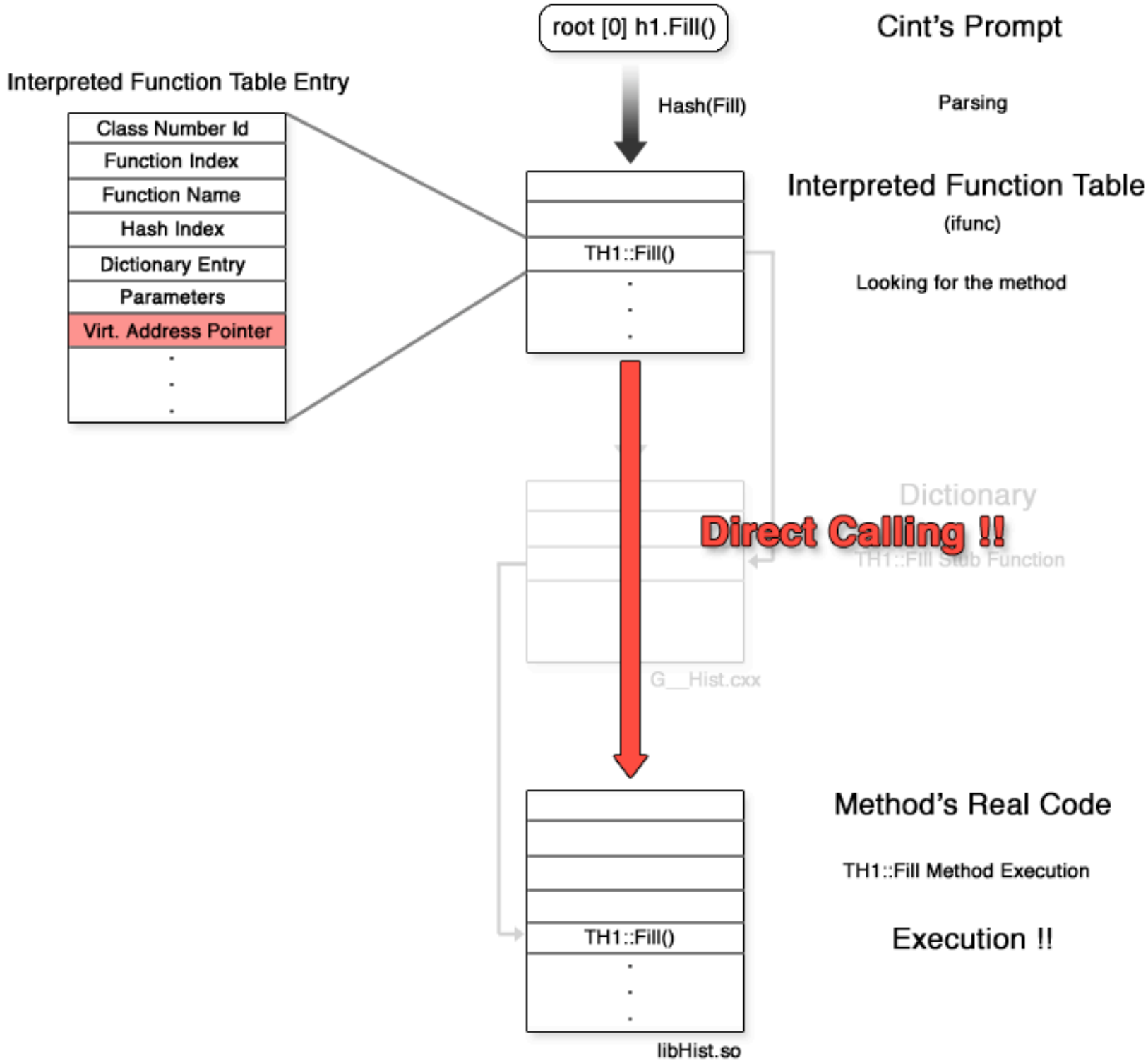
```
static int G__G__Hist_118_0_39(G__value* result7, G__CONST char* funcname, struct G__param* libp, int hash)
{
   G__letint(result7, 105, (long) ((TH1*) G__getstructoffset())->Fill((Double_t) G__double(libp->para[0])));
   return(1 || funcname || hash || result7 || libp);
}
```

Replace function pointers with mangled names

```
G__memfunc_setup2("Fill",391,"_ZN3TH14FillEd",(G__InterfaceMethod) NULL, 105, -1, G__defined_typename("Int_t"),
                  0, 1, 1, 1, 0, "d - 'Double_t' 0 - x", (char*)NULL, (void*) NULL, 80);
```

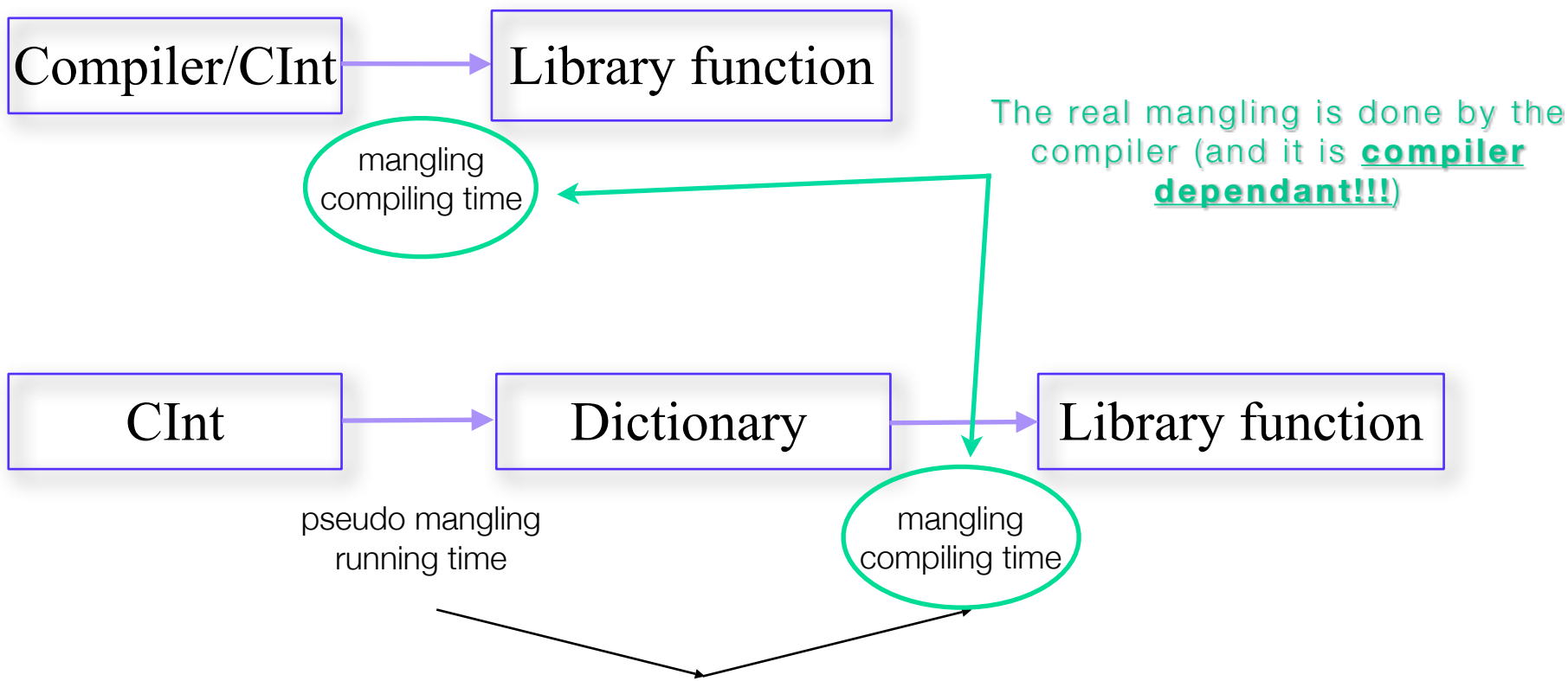
```
static int G__G__Hist_118_0_39(G__value* result7, G__CONST char* funcname, struct G__param* libp, int hash)
{
   G__letint(result7, 105, (long) ((TH1*) G__getstructoffset())->Fill((Double_t) G__double(libp->para[0])));
   return(1 || funcname || hash || result7 || libp);
}
```

Replace stubs with dynamic calls


root [0] h1.Fill()
Cint's Prompt
Interpreted Function Table Entry
Class Number Id
Function Index
Function Name
Hash Index
Dictionary Entry
Parameters
Virt. Address Pointer
Hash(Fill)
Parsing
TH1::Fill()
Interpreted Function Table
(ifunc)
Looking for the method
Dictionary
Direct Calling !!
TH1::Fill Stub Function
G__Hist.cxx
Method's Real Code
TH1::Fill Method Execution
TH1::Fill()
Execution !!
libHist.so

Finding the mangled name of a function

We need the mangled name in order to call a function

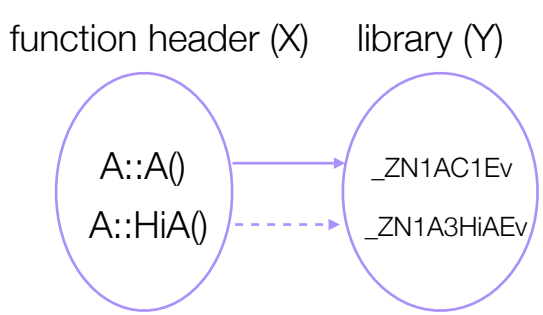


The dictionary could be seen as a bijective function that maps c++ function declarations to a certain string (string which will be associated to the symbol by the compiler)

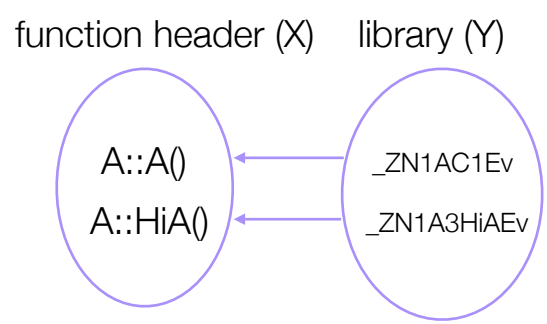
Finding the mangled name of a function

- The idea is to avoid that layer of indirection.
  - We still **don't** know how to do the mangling.
- But we know how to do the demangling (or at least, we know who to call to do it ;)).

Instead of going from set X to set Y
for a given x in X



Go from set Y to set X
for all y in Y



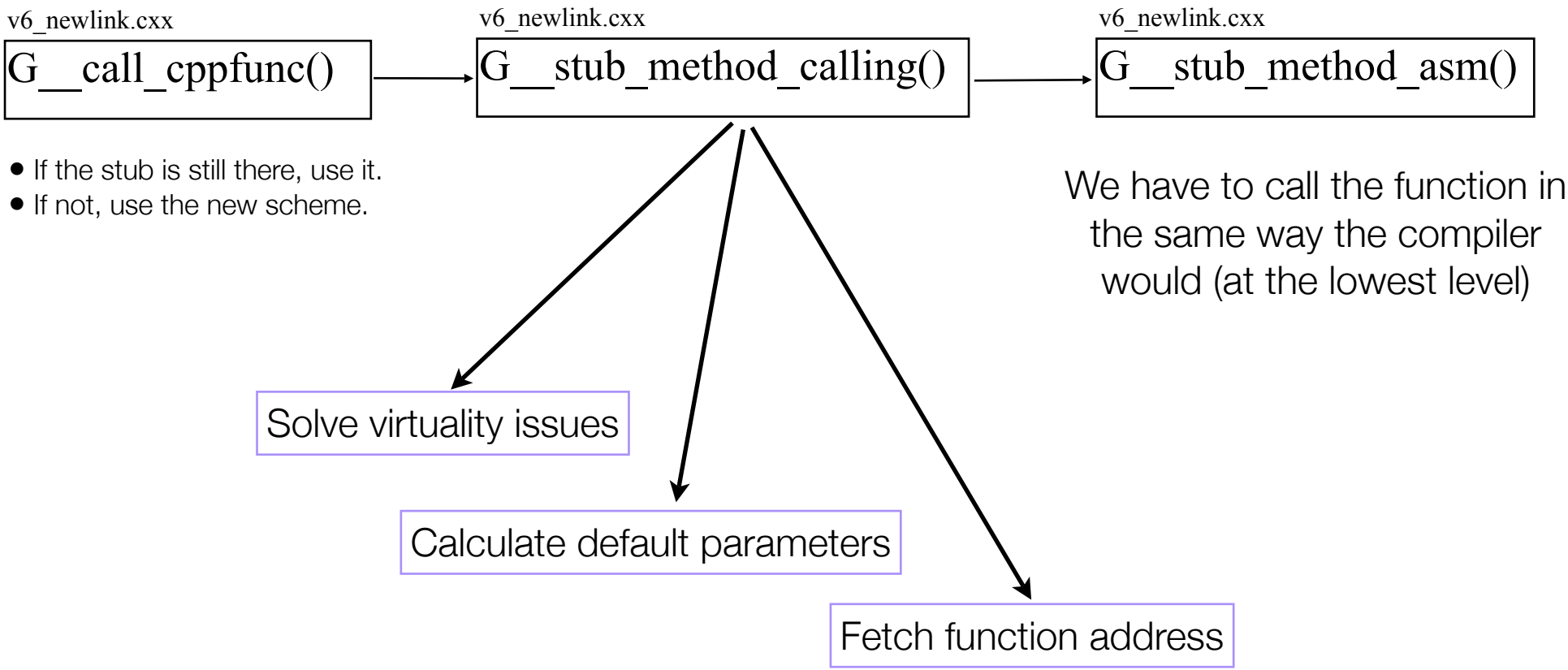
Finding the mangled name of a function

- This approach writes in stone the biggest side effect:
  - We will need to demangle ALL the symbols in a file just to find the name of 1 function.

- The demangling process might not be too expensive but what happens when we have thousands and thousands of symbols in a library?
  - libCore.so has 27.000 symbols
  - The set of ROOT libraries has 300.000
  - Mangled names are 46 char long
  - Demangled names are 27 char long

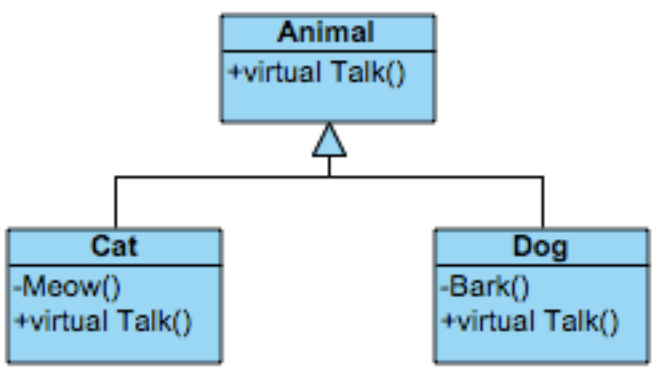
- This is less important if we move this procedure to compiling time (dictionary generation).

Direct function call: Big picture



Virtual functions: Almost impossible to get right.

Do you meow or bark?




```
Animal *animal = new Animal();
Cat    *cat    = new Cat();
Dog    *dog    = new Dog();

animal->Talk();
cat->Talk();
dog->Talk();
```

```
I shouldn't talk because I'm a poor animal without soul
I'm a cat... I do Meow Meow
I'm a dog... I do Woff Woff
```

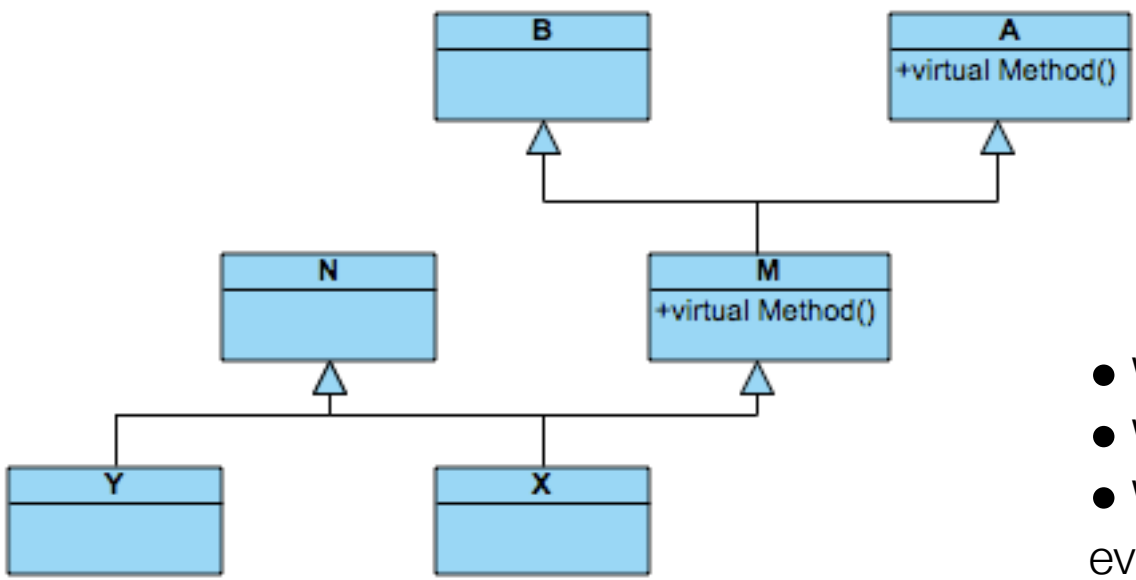
```
Animal *aniDog = new Dog();
Animal *aniCat = new Cat();

aniCat->Talk();
aniDog->Talk();
```

```
I'm a cat... I do Meow Meow
I'm a dog... I do Woff Woff
```

Notes:

- C++ can tell dogs and cats apart.
- For an “aniCat” CInt won't be aware that it's a cat, it will just pass the animal to c++.
- Without the stub function it's our problem to differentiate between dogs and cats.

Virtual functions: A complex implementation




```
A* a_a = new A();
a_a->Method();

A* a_m = new M();
a_m->Method();

A* a_x = new X();
a_x->Method();
```

- When do we call **A::Method()** ?
- When do we call **M::Method()** ?
- What is the **this** pointer passed to every function?

Not only do we have to find the right method in the hierarchy, we also have to pass the right **this** pointer.

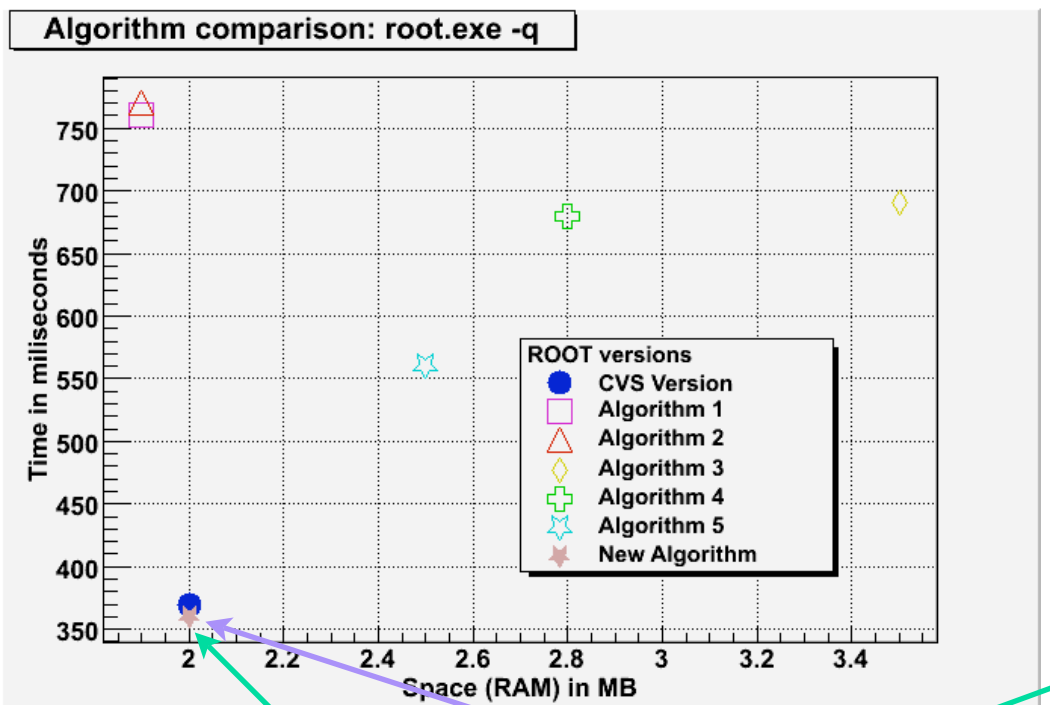
- This is very tricky and is the equivalent of implementing something known as **virtual function table** or **virtual dispatch** in compiler lingo.

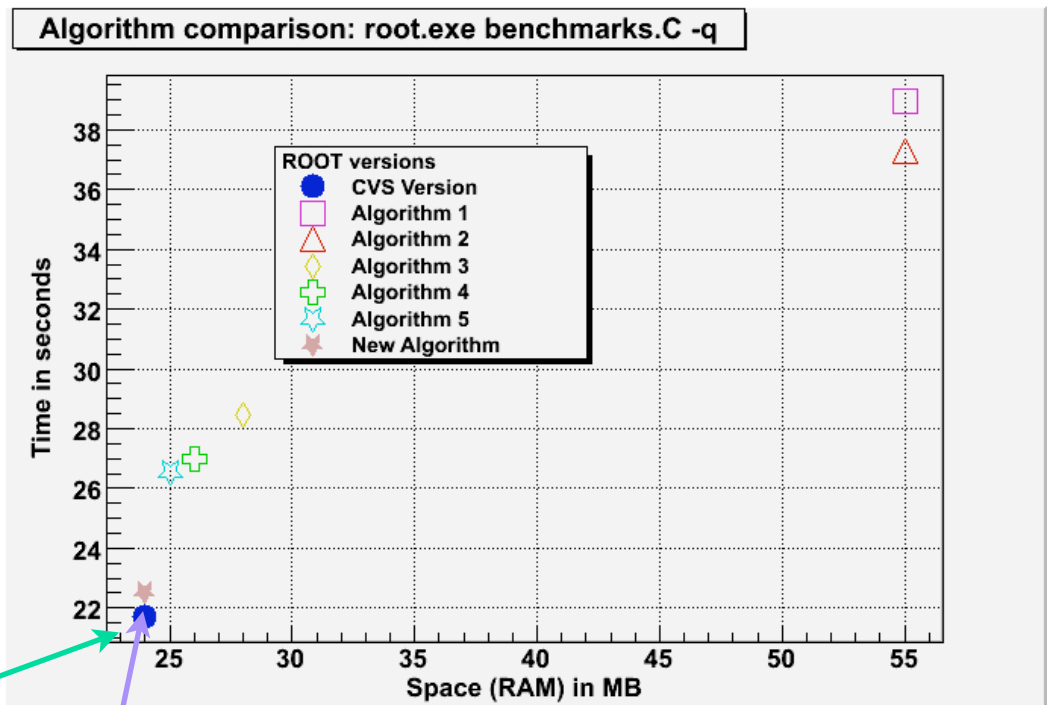
Results: How much do we gain?

- Removing the stub functions from the dictionaries source:
  - Version 5.17: 50.37MB
  - Actual status: 30.09MB

    (-20.28 MB, **40.2%**)
- If the dictionaries represent 52% of the total size of the libraries then we are reducing these libraries by 20%.

Results: What is the price to pay? (time or memory)

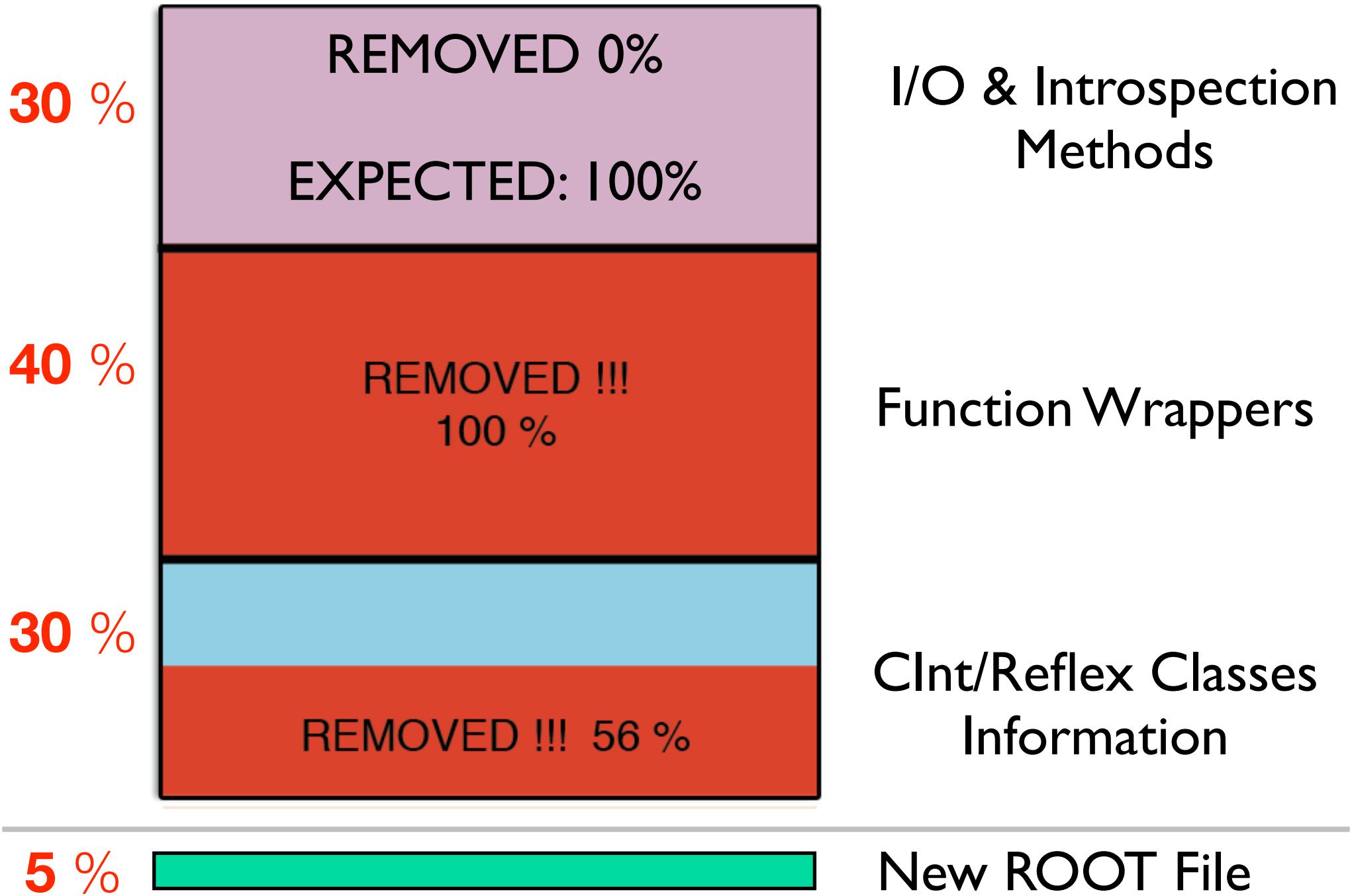




Memory consumption (X axis) hasn't been changed !!

Performance is constant at startup but decreases by 4% running the benchmark

Conclusions



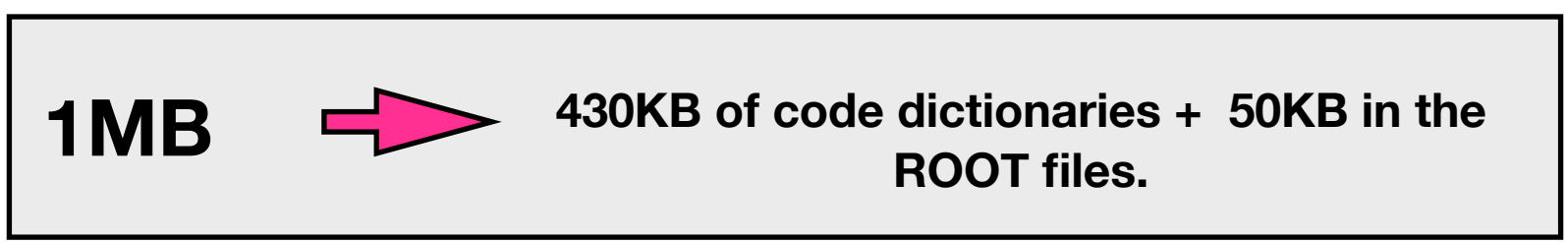
Conclusions

- Our goal is to reduce the code needed in the dictionaries as much as possible.

- A big effort has been done to move the class descriptions to ROOT files and eliminate the stub functions.

  - For instance, if we have a set of classes of 1MB, we needed an additional 1MB for the dictionaries. Now instead of 1MB of code dictionaries, we have:

Current Status



- We still need to get rid of the reflection information (Shadow classes, ShowMembers(), etc), which represent almost **30%** of the old dictionaries.

Expected