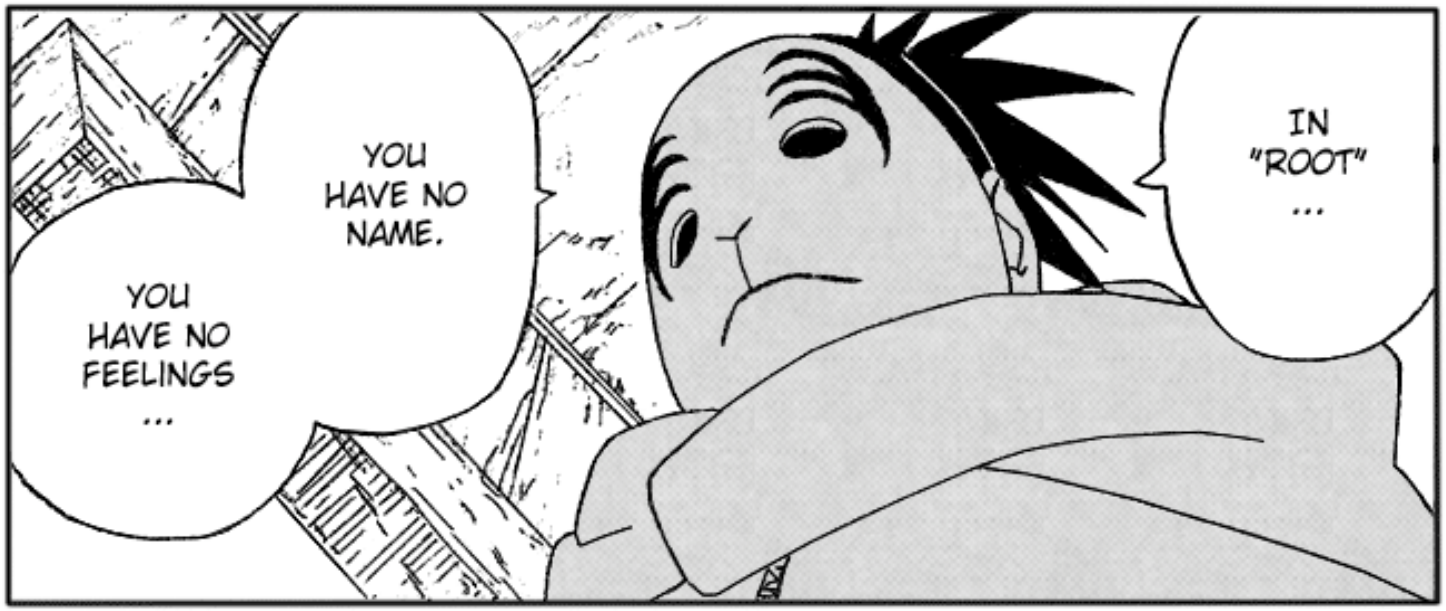


Getting rid of the “G__” thingies

ROOT Team Meeting
CERN

Diego Marcos && Leandro Franco

Road Map

- Currents Status
  - Actual use of the dictionaries.
- Proposal
  - Removal of stub functions in the dictionaries.
  - Needed changes
- Advantages
  - Dictionary size reduction (and its consequences).
- Problems

Current status

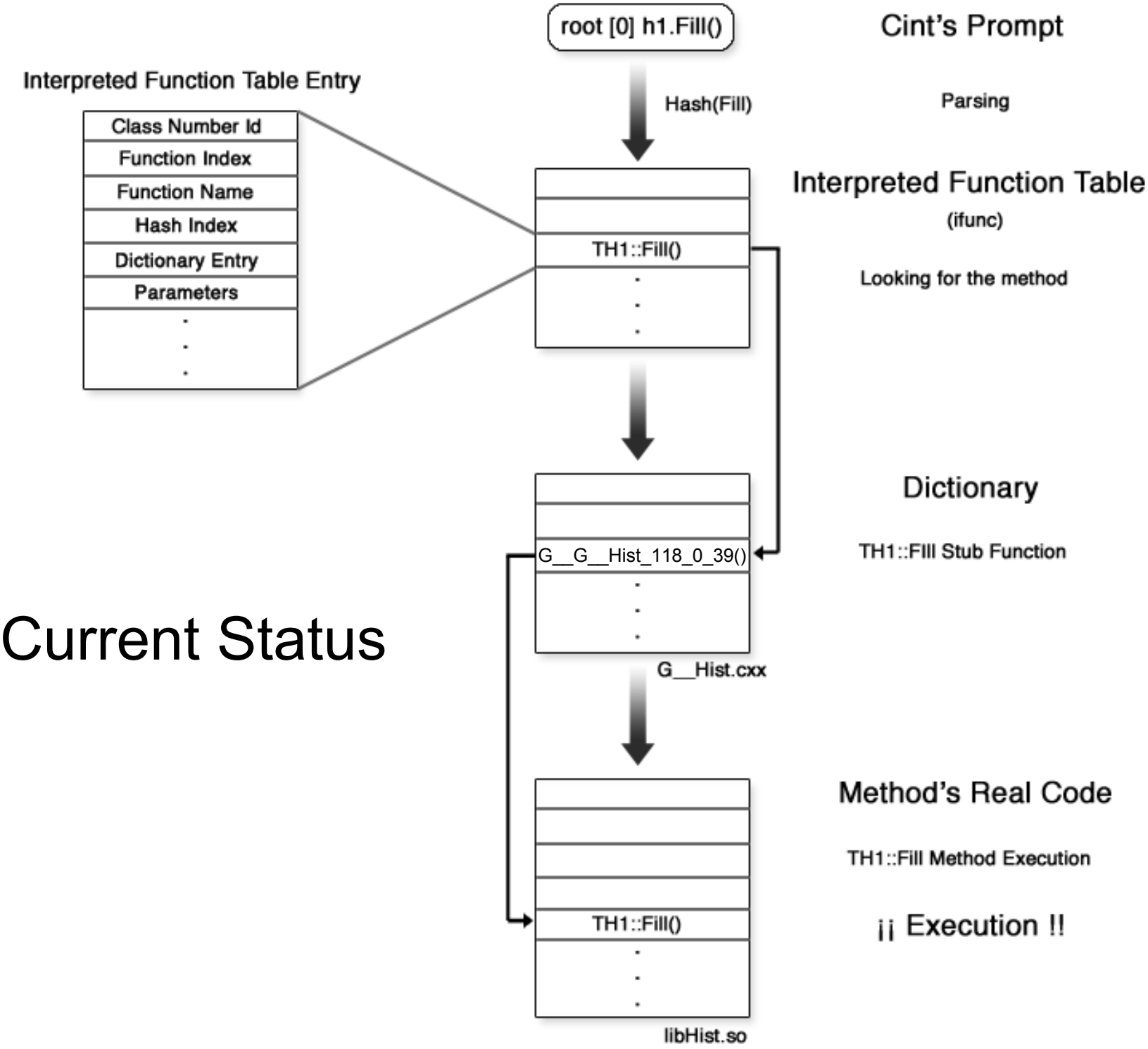
- Initializing CInt Data structures.
  - At loading time we have one “G__memfunc_setup” call for each stub function (in a file like G__Hist.cxx).
  - This function creates a new field in the G__ifunc_table with attributes like name, hash, type, parameters, etc. (Including the pointer to the stub function!)

```
G__memfunc_setup("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 function in the dictionary... for every single method!!!

Current Status



What is a function call in CInt

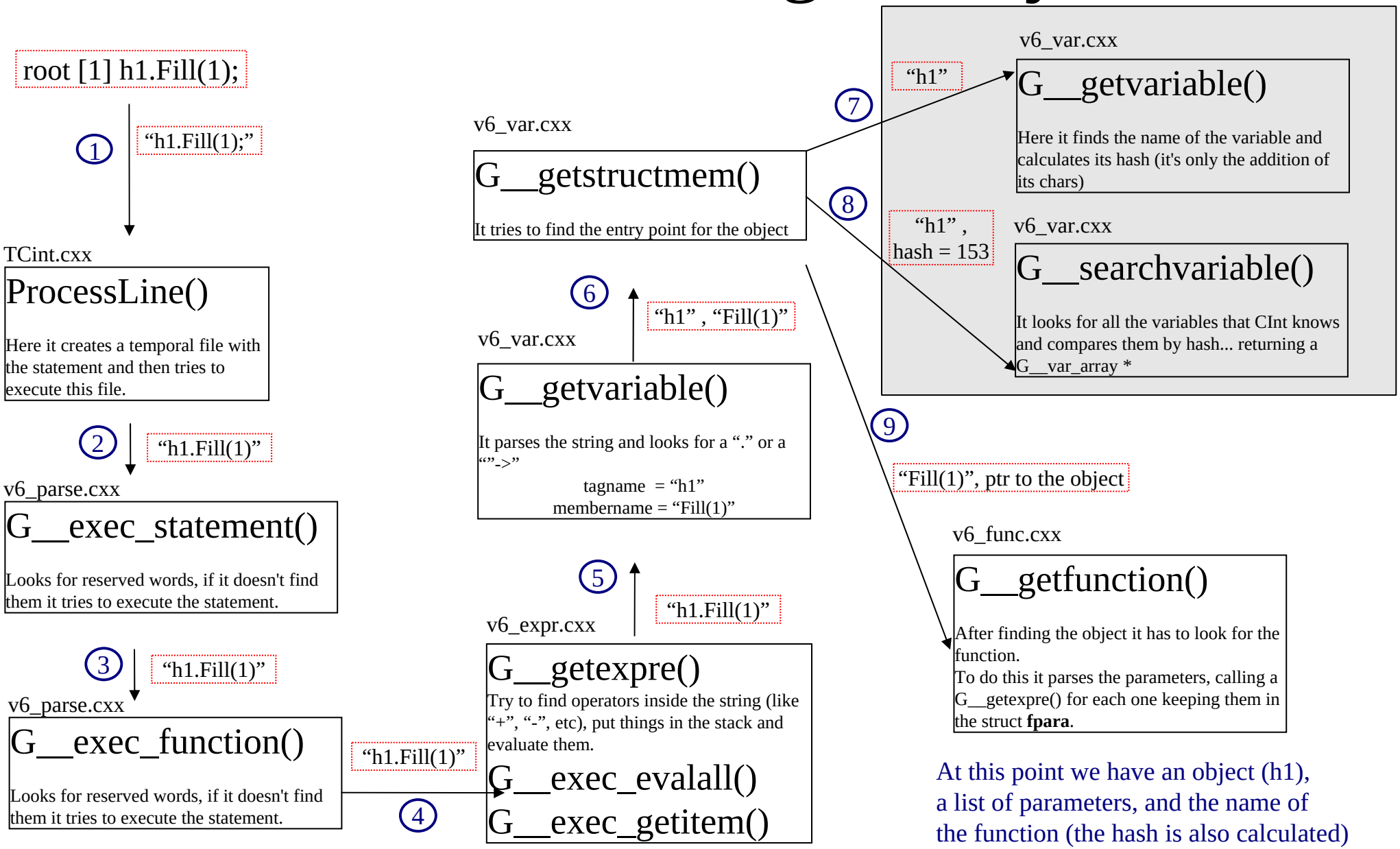
- A CInt call is something similar to a C/C++ statement that is usually executed from ROOT

```
-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

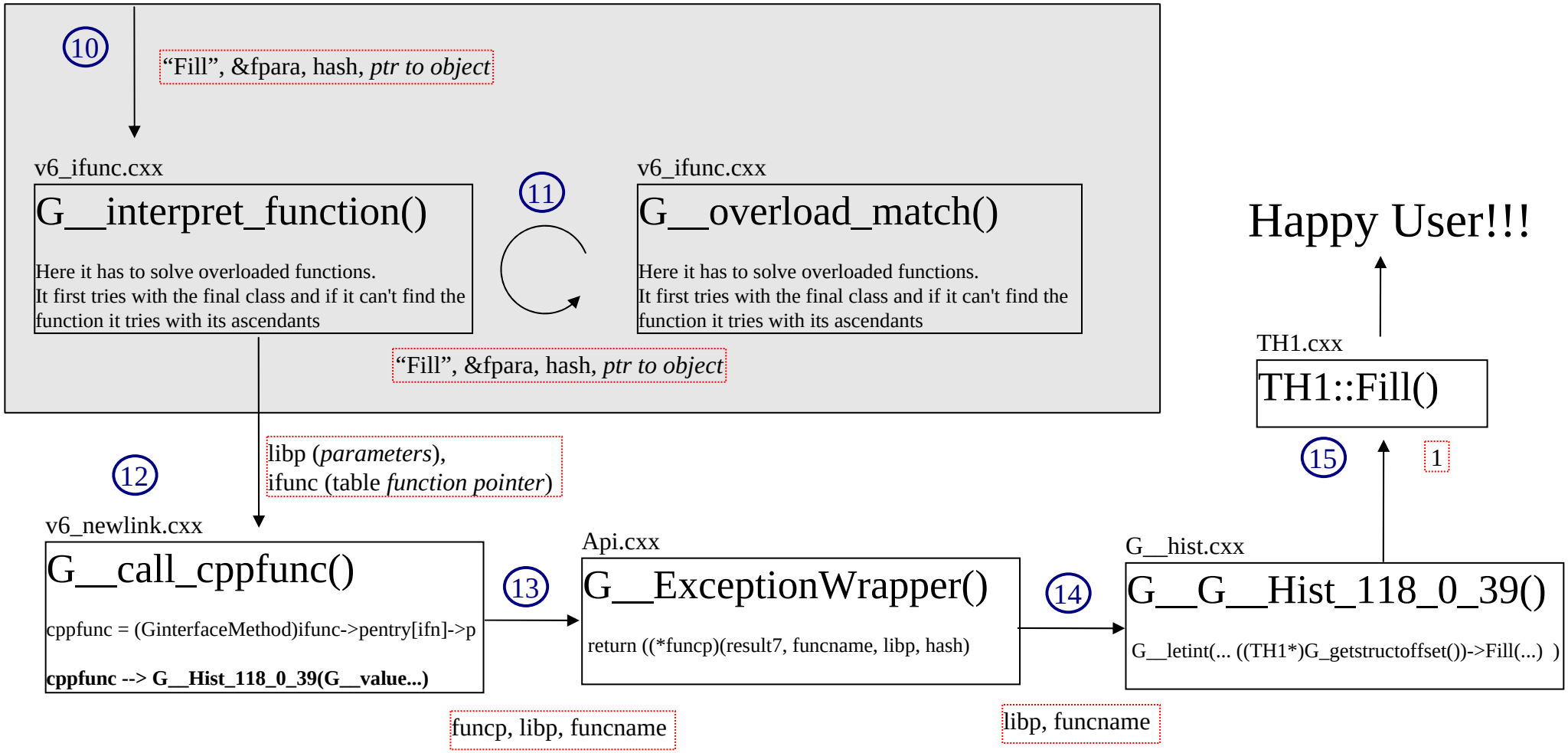

```
root [1] h1.Fill(1);
```

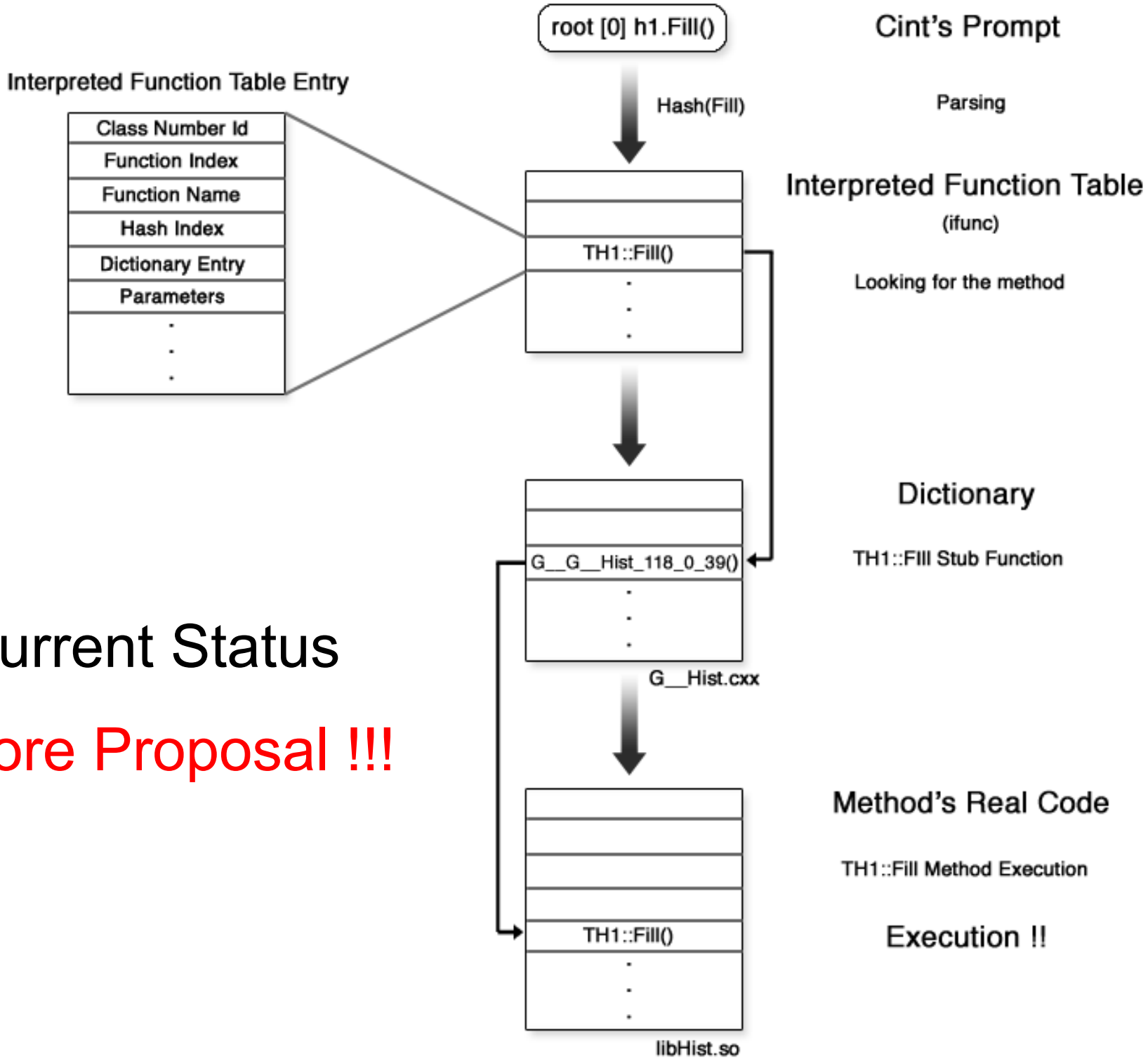
Now... the long history



Almost there.....

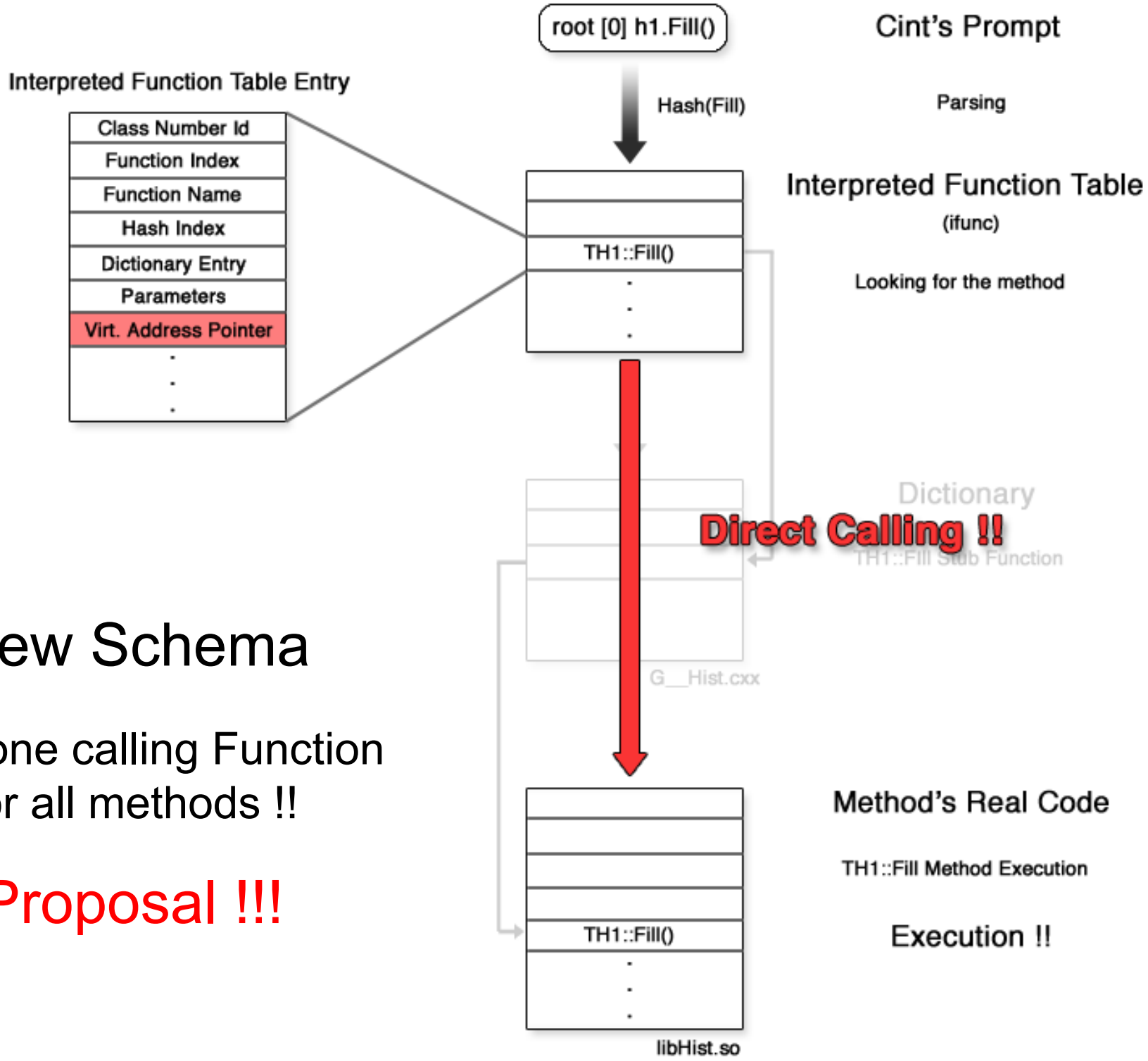
At this point we have an object (h1),
a list of parameters, and the name of
the function (the hash is also calculated)





Current Status

Before Proposal !!!

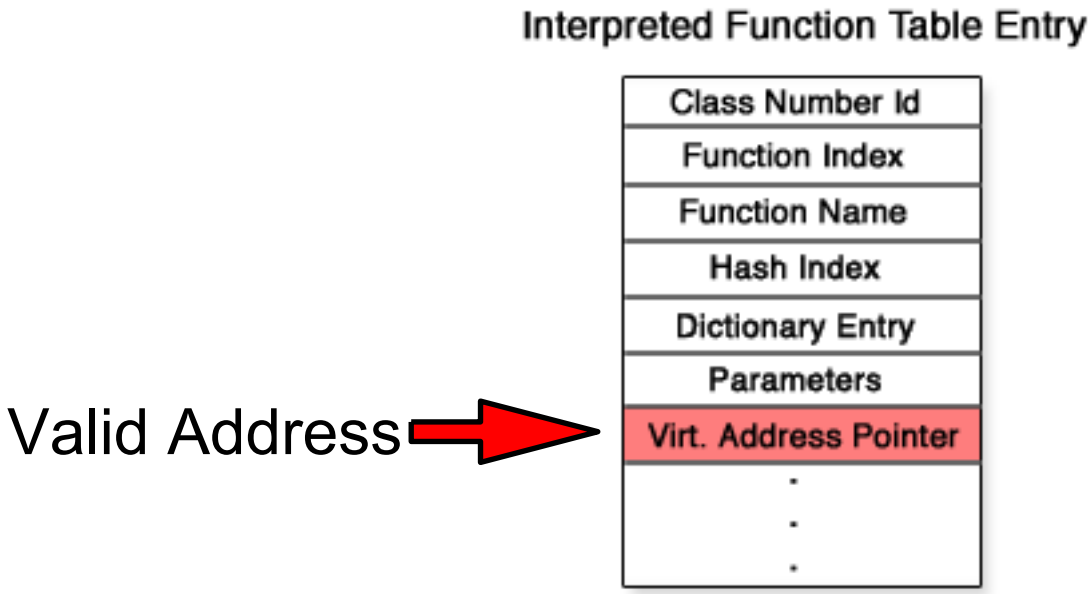


New Schema

Only one calling Function
for all methods !!

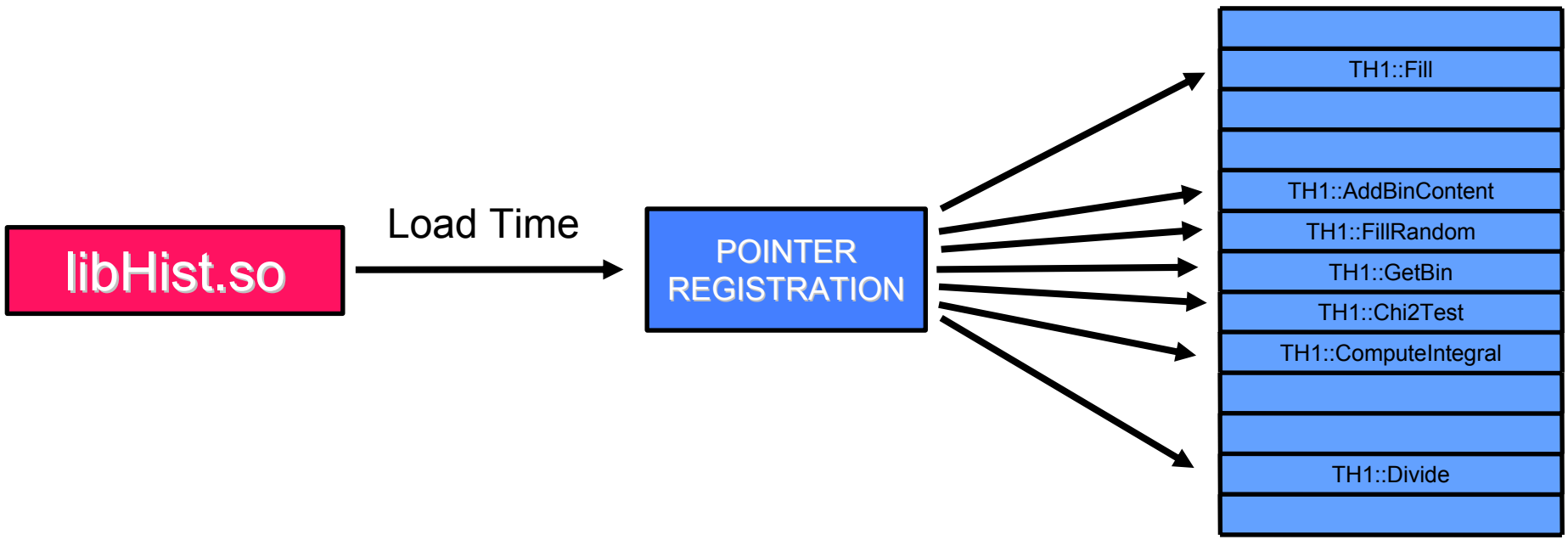
Proposal !!!

Structure Initialization



Structure Initialization - Two Strategies

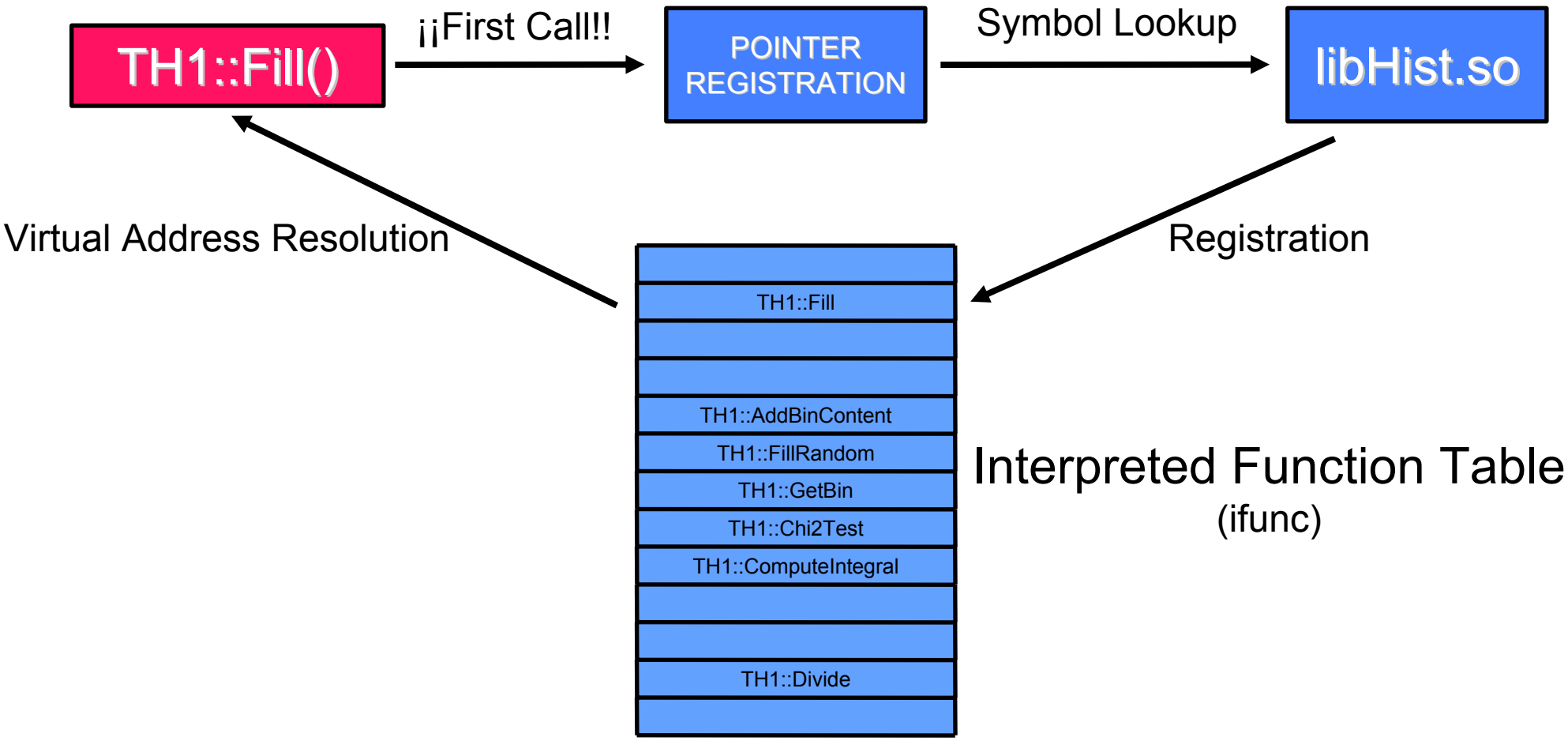
"Apriori" Initialization



- All Pointers for all symbols in the loaded library
- Increasing of load time
- We register pointers that we will not reference

Structure Initialization - Two Strategies

“Lazy” Initialization - On Demand Initialization



Structure Initialization - Two Strategies

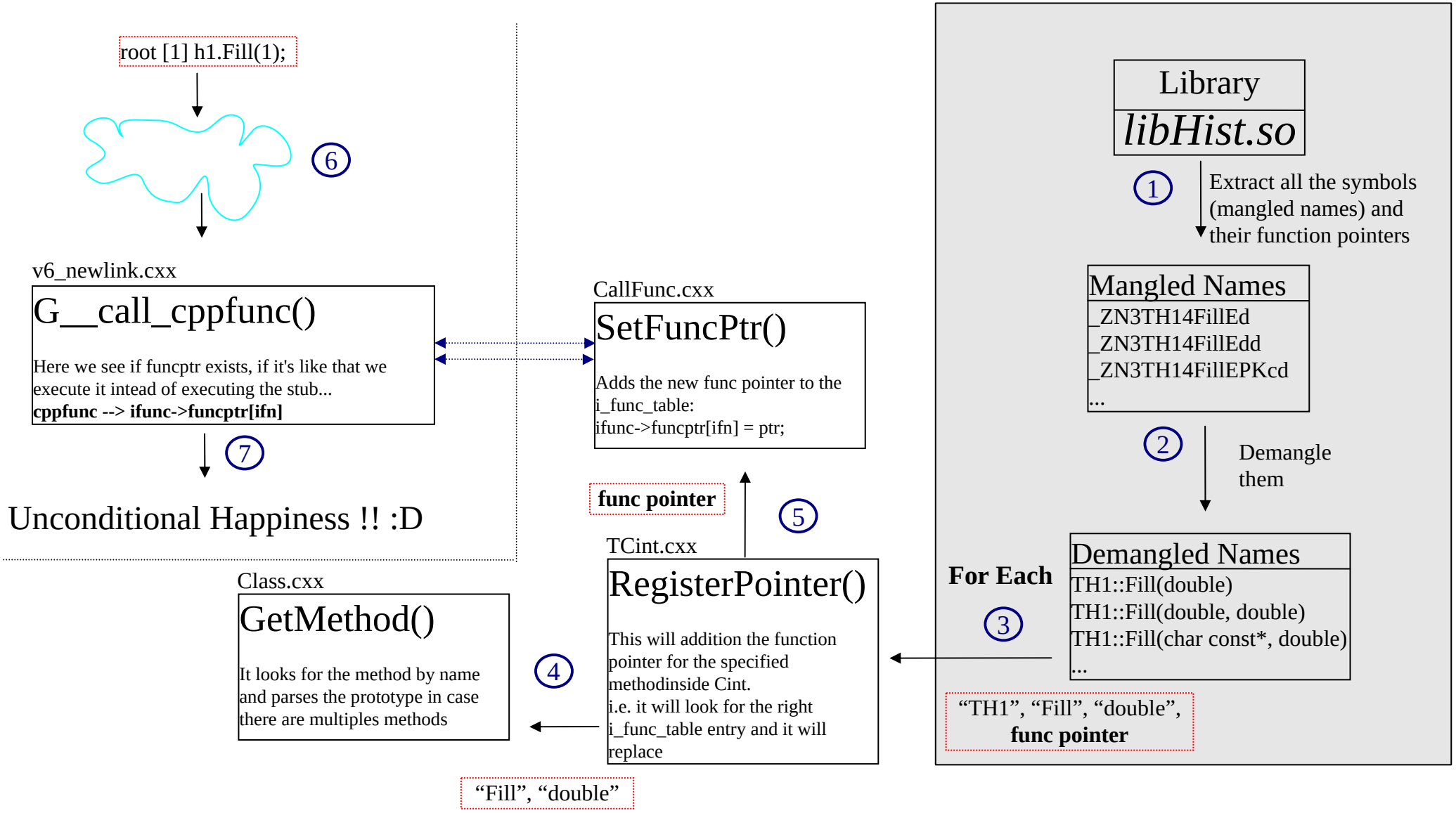
“Lazy” Initialization - Drawbacks

LibHist.so exports 4600 symbols
Average Symbol Length = 27 Characters
Symbol Lookup = String Comparisons

“Lazy” Initialization - Benefits

Speed Up - Libraries Load Time

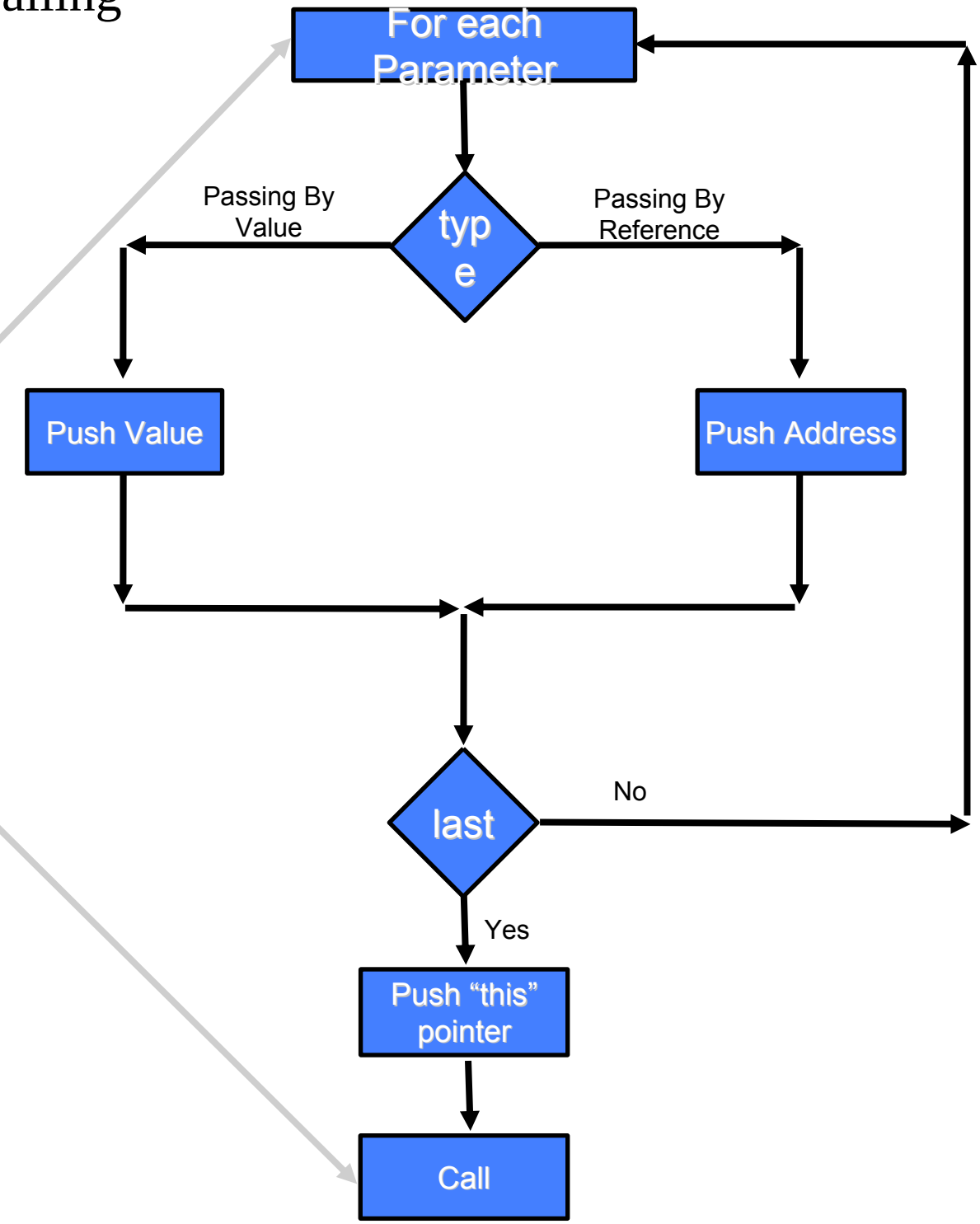
Proposal


root [1] h1.Fill(1);
6
v6_newlink.cxx
G__call_cppfunc()
Here we see if funcptr exists, if it's like that we execute it intead of executing the stub...
cppfunc --> ifunc->funcptr[ifn]
7
Unconditional Happiness !! :D
CallFunc.cxx
SetFuncPtr()
Adds the new func pointer to the i_func_table:
ifunc->funcptr[ifn] = ptr;
func pointer
5
TCint.cxx
RegisterPointer()
This will addition the function pointer for the specified methodinside Cint.
i.e. it will look for the right i_func_table entry and it will replace
4
Class.cxx
GetMethod()
It looks for the method by name and parses the prototype in case there are multiples methods
"Fill", "double"
Library
libHist.so
1
Extract all the symbols (mangled names) and their function pointers
Mangled Names
_ZN3TH14FillEd
_ZN3TH14FillEdd
_ZN3TH14FillEPKcd
...
2
Demangle them
Demangled Names
TH1::Fill(double)
TH1::Fill(double, double)
TH1::Fill(char const*, double)
...
For Each
3
"TH1", "Fill", "double",
func pointer

Intel386 Assembler Direct Calling

Interpreted Function Table Entry

Class Number Id
Function Index
Function Name
Hash Index
Dictionary Entry
Parameters
Virt. Address Pointer
.
.
.



Intel386 Method Calling Examples

TH1::Fill(Double_t x)

```
push "this pointer"
push "x value"
call "Fill"
```

TH1::Fill(const char * name, Double_t x)

```
push "this pointer"
push "x value"
push "name address"
call "Fill"
```

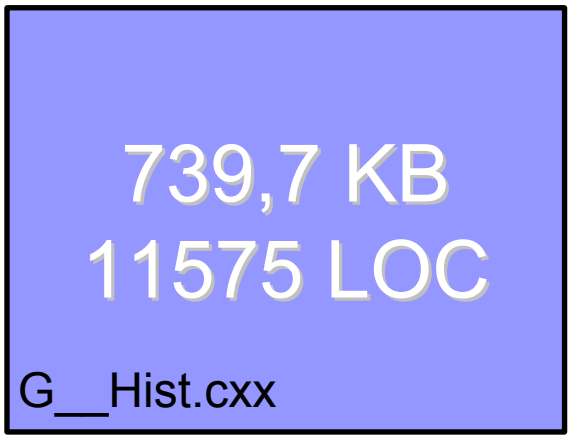
¡¡We have to adapt this calling schema for each processor family!!

Source Code's Size reduction

Example: LibHist.so Dictionary

Current Status

Future Status



Reduction Factor = 2,15

CONSEQUENCES

- Compilation Time Decreasing
- Disk Space Saving
- Memory Use Saving

First problem

- Mangling vs Demangling
  - Mangling is direct (From the function prototype we can get the function pointer)
  - But there is no way to do the mangling in a portable way.
  - Demangling is indirect (Demangle all and do a lookup when a function is needed).
  - Some effort has been made to have a common demangling scheme.

Second (big) problem... virtuality

Based on the Virtual Table Layout from the ABI:

- Category 0: Trivial
  - No virtual base classes.
  - No virtual functions.
- Category 1: Leaf
  - No inherited virtual functions.
  - No virtual base classes.
  - Declares virtual functions.

Second (big) problem... virtuality

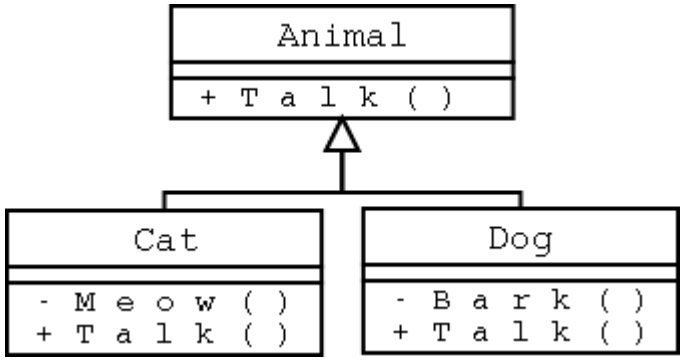
- Category 2: Non-Virtual Bases Only
  - Only non-virtual proper base classes.
  - Inherits virtual functions.
- Category 3: Virtual Bases Only
  - Only virtual base classes (but those may have non-virtual bases).
  - The virtual base classes are neither empty nor nearly empty.
- Category 4: Complex

Virtuality

- Categories 0: Trivial
  - We have the pointer to a function (in the i_func_table) and a pointer to the object so we “just” call the function passing the object as the this parameter.
- Category 1: Leaf
  - Idem... for the moment we don't need the virtual table.

Virtuality

- Category 2: Non-Virtual Bases Only
  - 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();
```

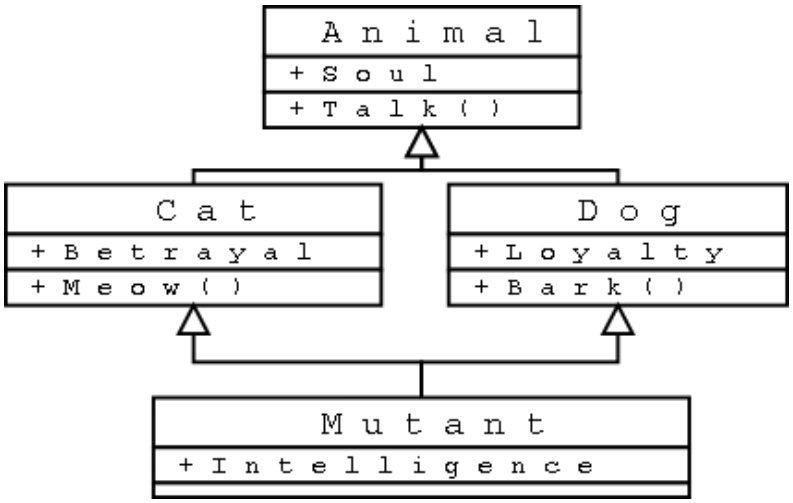
```
I'm a cat... I do Meow Meow
I'm a dog... I do Woff Woff
```

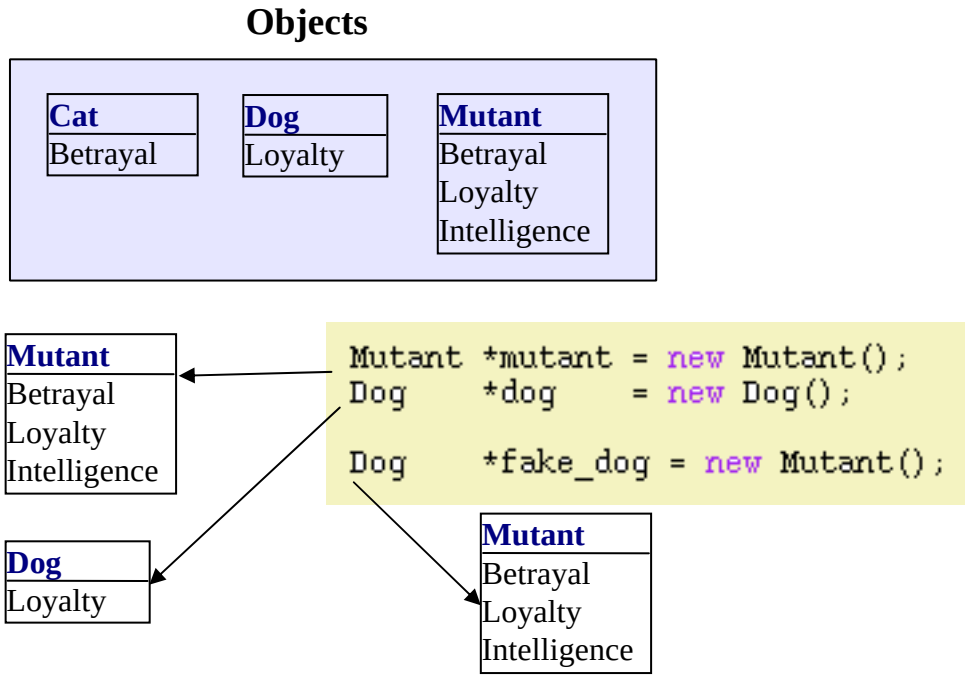
Notes:

- 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.

Virtuality

- Category 3: Virtual Bases Only
  - If you are a mutant... can you really bark?





- A pointer to a mutant points to the beginning of the object.
- In a call mutant->Bark(), c++ can not pass *this* pointing to "betrayal" because a dog knows nothing about it.
- But without the stubs functions we get a pointer to a Mutant and we have to force it to bark!!!... i.e. turn it to a dog even if it doesn't feel like it.

Virtuality

- Category 4: Complex
  - Our life is sad enough as it is now :(... we will get back at it after dealing with the other issues.