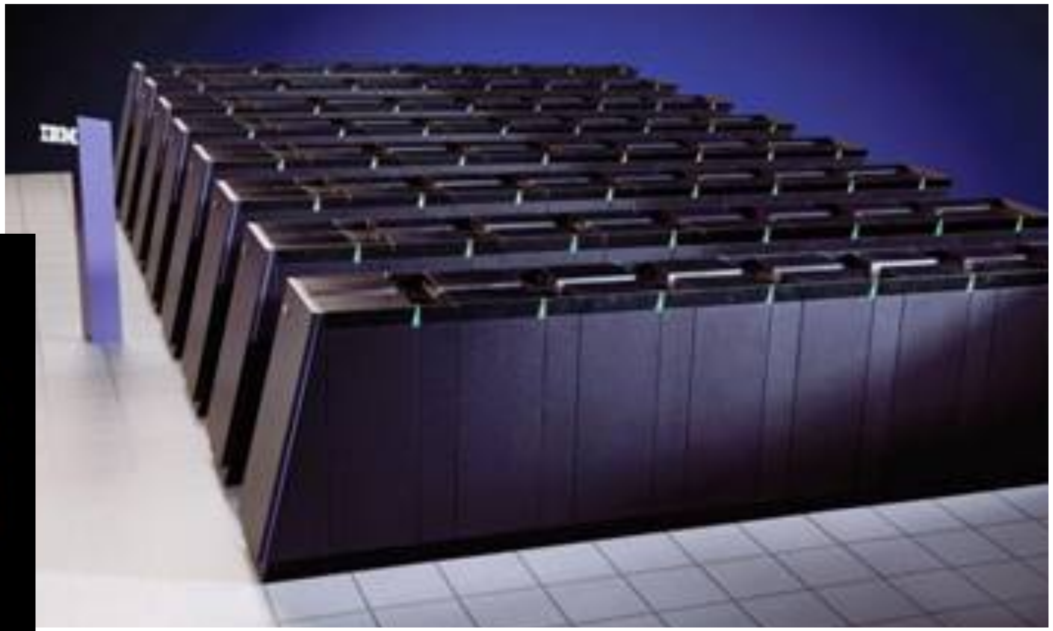
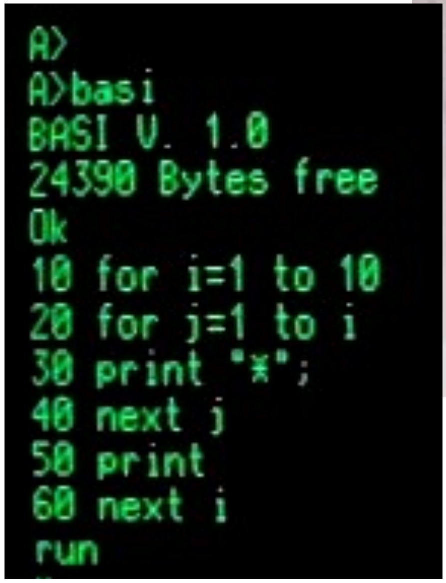
The Role of Interpreters in High Energy Physics





```
A>
A>basi
BASI V. 1.0
24390 Bytes free
Ok
10 for i=1 to 10
20 for j=1 to i
30 print "*";
40 next j
50 print
60 next i
run
```

VEESC 2010

Philippe Canal (Fermilab, Chicago, IL)

High Energy Physics

Large datasets

- 15 petabytes a year

Often analyzed (directly or indirectly)

- more than half a petabytes is reprocessed per day in just the Open Science Grid!

Using up a lot of cpu

- More than 16 millions cpu hours a month on OSG.

Every little bit can make a big difference.

High Energy Physics

Thousands of collaborators. Each physicist is a developer. Participation and CS skill varies.

- Framework
  - Reconstruction, Simulation
  - Modules (some common, some not)
  - Run on large scale data set

- Analysis (private or shared).
  - Run on smaller scale data set
  - Shared by small(er) groups.
  - Often but not always relies on the framework.

Common threads: data formats, core tools (ROOT/Cint/PyRoot).

Interpreter Applications

Wide Range:

- Job Management, submission, error control
- Gluing programs and configurations
- "Volatile" algorithms subject to change or part of configuration

In use in various forms for decades:

  - Kumacs (adhoc), Comis (Fortran interpreter), 1980s
  - CINT (C++ interpreter), 1990s
  - perl, bash, tcsh, Tcl/Tk, Python, etc.

CINT

Started in 1991 by Masaharu Goto, originally in C.

>300k *real* LOC (excluding comments / empty lines)

Default interface to ROOT (data analysis framework used by 20k users worldwide)

- C++ Parser
- Dictionary generator
- Reflection data manager
- Code and library manager
- C++ Interpreter

Non Intrusive Input/Output Framework with automatic schema evolution

From Text

Analyses subject to change

- Different cuts, parameters
- Different input / output

Configure with ease using text files:

```
JetETMin: 12
NJetsMin: 2
```

```
<JetETMin value="12"/>
<NJetsMin value="2"/>
```

To Code

Volatile Algorithms:

Changes to algorithms themselves, especially during development:

» two jets and one muon each

» three jets and two muons anywhere

» no isolated muon

```
TriggerFlags.doMuon=False
EFMissingET_Met.Tools = \
   [EFMissingETFromFEBHeader()]
```

Configuration not trivial!

Algorithms as Configuration

Acknowledge physicists' reality:

- Refining analyses is asymptotic process
- Programs and algorithms change
- Often tens or hundreds of optimization steps before target algorithm is found

- **Almost** the same:
  - » background analysis vs. signal analysis
  - » trigger A vs. trigger B

Interpreter Advantage: Data Access

- **Make it easier to use higher level constructs**
- Hide data details irrelevant for analysis

  `vector` – `hash_map` – `list`? Who cares!

```
foreach electron {...
```

- Framework provides job setup transparently

```
MyAnalysis(const Event& event)
```

- Remove (*hide*) compilation step
- (Often) Simplify memory management

Interpreter Advantage: Localized

Compiled: distributed changes

usually many packages need changes by *regular physicists* as opposed to release managers

Interpreter: localized changes

- Easier to track (CVS / SVN)
- Less side effects
- Feeling of control over software
- Eases communication / validation of algorithms

Interpreter Advantage: Agility

Interpreter boosts users' agility compared to configuration file:

- more expressiveness
- thus higher threshold for recompilation of the framework

Distribution is simplified

- One package for all platforms
- But: when more advanced features and packages are used the deployment becomes more difficult.

Compiled vs. Interpreter

Compiled:

usually many packages need changes by *regular physicists* as opposed to release managers

Interpreter:

helps localize changes,

modular algorithmic test bed

Why Not To Use Interpreters?

Slower than compiled code

Difficult to quantify:

- nested loops `foreach event { foreach muon {...`
- calls into libraries `hist.Draw()`
- virtual functions, etc.

In our experience usually O(1)-O(10) slower than compiled code

Interpreters *can not* replace compiled code for the core components and cpu intensive algorithm

Why Not To Use Interpreters?

- Slower than compiled code
- Not integrated well with reconstruction software
- Seen as unreliable
- Not part of the build system
- Difficult to debug
- Lack of static type checks

Where Not To Use Interpreters?

Interpreters ***can not*** replace compiled code for the core components and cpu intensive algorithms:

- Input/Output, Minimization
- Trackings, Simulations, Jet clustering algorithms, etc.

Dynamically typed languages are inherently slower that statically typed language:

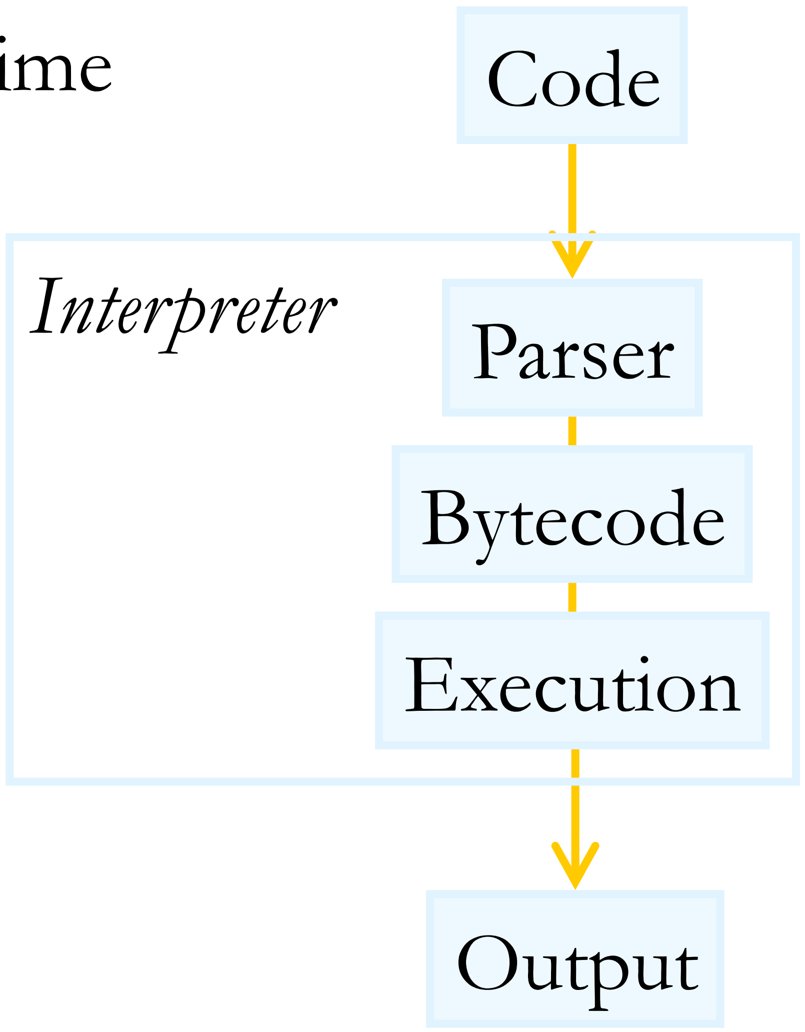
- at the very least due to the need to check the type.

Consequently:

- Any interpreter needs to interface with compiled code.

Ideal Interpreter

1. Fast, e.g. compile just-in-time
2. No errors introduced: quality of all ingredients
3. Good support for using and accessing user provided compiled code libraries.



Ideal Interpreter

4. Smooth transition to compiled code, with compiler or conversion to compiled language
5. Straight-forward use: known / easy language.
6. Possible extensions with conversion to e.g. C++

```
foreach electron in tree.Electrons
```

```
vector<Electron>* ve = 0;
tree->SetBranchAddress("Electrons", ve);
for (int i=0; i<ve.size(); ++i) {
  Electron* electron = ve[i];
```

Interpreter Options: Custom

Even though not interpreted as interpreter:

Parameters

```
postzerojets.nJetsMin: 0
postzerojets.nJetsMax: 0

+postZeroJets.Run: NJetsCut(postzerojets) \
                   VJetsPlots(postZeroJetPlots)

postzerojets.JetBranch: %{VJets.GoodJet_Branch}
```

Algorithm

Interpreter Options: Python

- Distinct interpreter language
- Interface to ROOT
- Rigid style
- Easy to learn, read, communicate

```
h1f = TH1F('h1f','Test',200,0,10)
h1f.SetFillColor(45)
h1f.FillRandom('sqroot', 10000)
h1f.Draw()
```

Python: Abstraction

Real power is abstraction:

- can do without types:

```
h1f = TH1F(...)
```

- can loop without knowing collection:

```
for event in events:
  muons = event.Muons
  for muon in muons:
    print muon.pt()
```

Major weakness:

compile time errors become runtime errors

Interfacing Challenges

Non-overlapping concepts

- Lifetime
  - Garbage collection vs. directed management.
  - Return values.

```
Owned* getOwned() {
   // Owner self-registers
   // in a list
   Owner* o = new Owner();
   return o->GetOwned();
}
```

```
def getOwned():
    o = Owner();
    return o.GetOwned()
o2 = getOwned()
# ouch, ~Owner() called
# destructing owner an owned
```

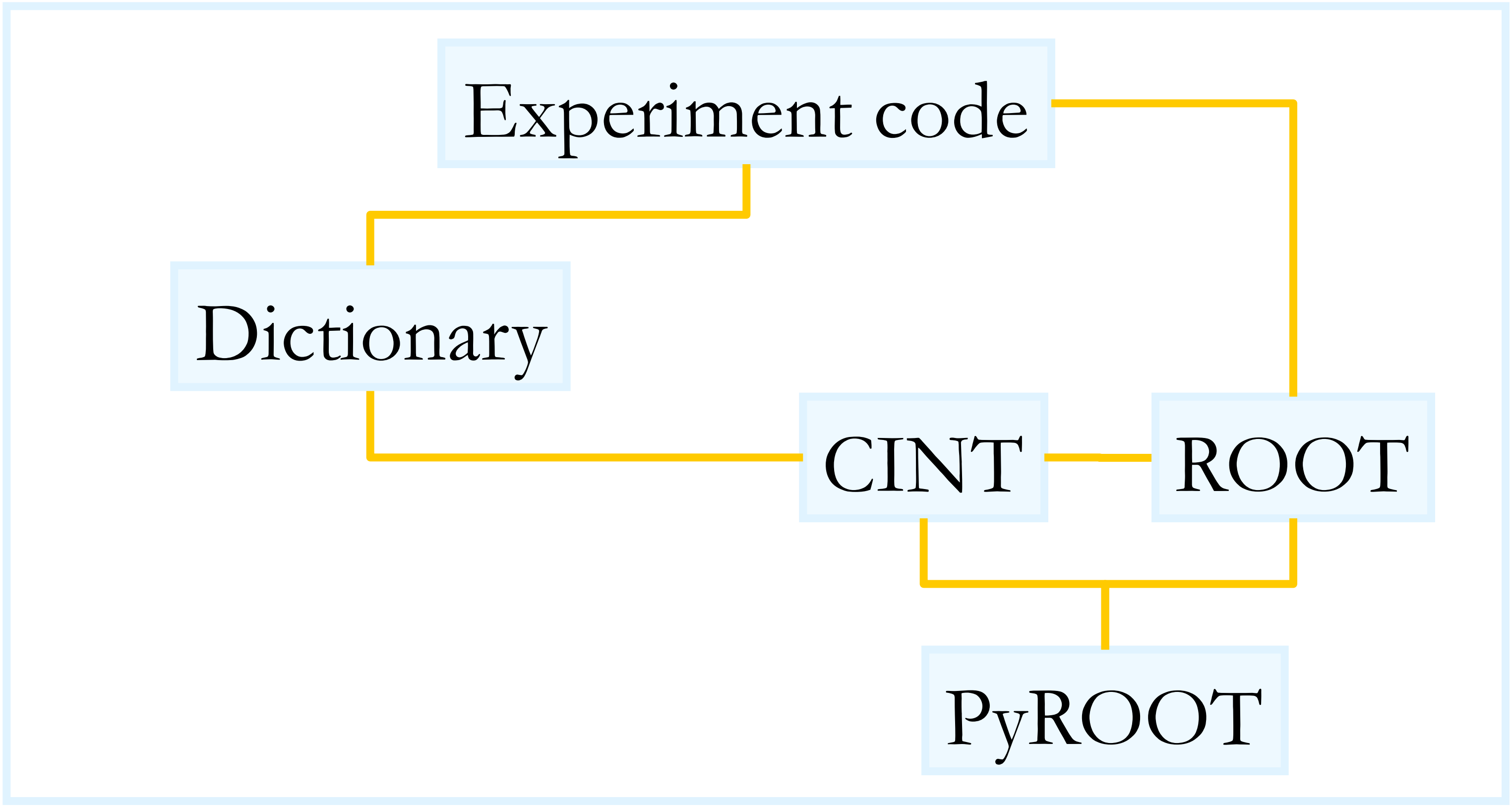
- Containers
- Template instantiation

Interfacing Challenges

- Creation of the interfacing wrappers
  - Can be automated at runtime if compiled language supports reflection and introspection.
  - Provided for C++ by CINT (see slide “CINT and Dictionaries)

PyROOT: The Maze

ROOT's python interface:



Common Interpreter Options: CINT

- C++ is prerequisite to data analysis anyway – interpreter often used for first steps
- Can migrate code to framework!
- Seamless integration with C++ software, e.g. ROOT itself
- Rapid edit/run cycles compared to framework

```
void draw() {
  TH1F* h1 = new TH1F(...);
  h1->Draw();
}
```

Common Interpreter Options: CINT

Forgiving

- automatic #includes, automatic library loading, can do without types

```
// load libHist.so
// #include "TH1.h"

void draw() {
  h1 = new TH1F(...);
  h1->Draw();
}
```

Common Interpreter Options: CINT

Covers large parts of ISO C++:
templates, virtual functions, etc.

>15 years of development!

Can be invoked from compiled code:

```
gROOT->ProcessLine("new Klass(12)");
```

Or from prompt, e.g. on a whole C++ file:

```
root [0] .L MyCode.cxx
```

CINT And Libraries

Call into library:

```
TH1F* h1 = new TH1F(...);
h1->Draw();
```

Even custom library:

```
root [0] gSystem->Load("Klass.so")
root [1] Klass* k = Klass::Gimme()
root [2] k->Say()
```

Knows what "Klass" is!

Translates "Klass::Gimme()" into a call!

CINT And Dictionaries

CINT must know available types, functions

- C++ does not provide this information at run-time.

Extracted by special CINT run from library's headers

- an alternative (Reflex) exists

Also prerequisite for data storage,
see "Data and C++" in backup slides.

Reflection and Dictionaries

Reflection is database, dictionary is data.

Refection data can be generated from

- user: `Reflect::AddClass("MyClass")`
- headers using modified compiler: GCCXML
- headers using custom parser: CINT
- *debug information*

Interpreter Access

Reflection allows interactive use of classes

```
gROOT->ProcessLine("A::f(1)")
```

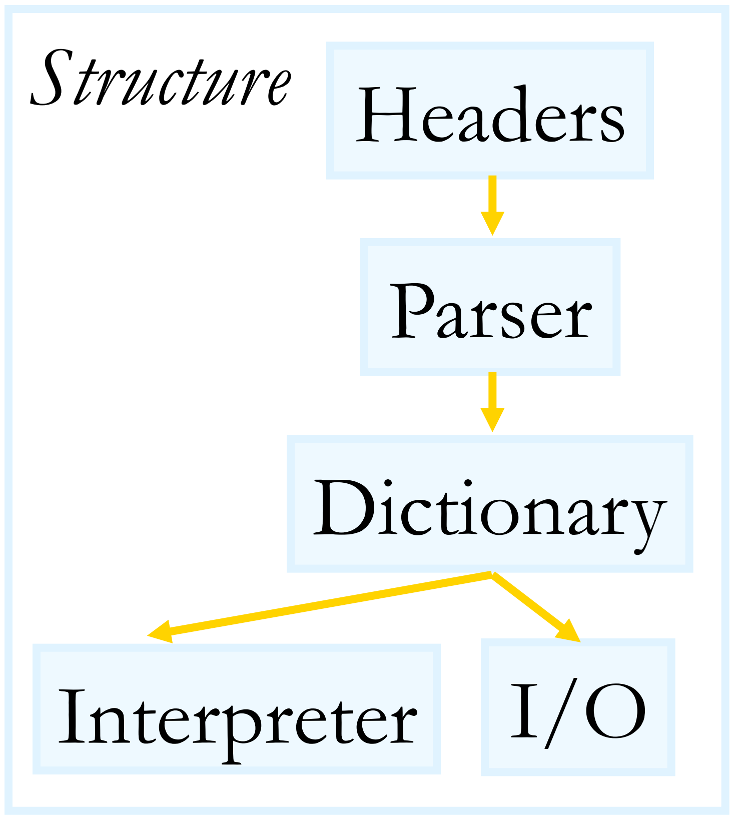
Interpreter knows type A, function f(), e.g.:

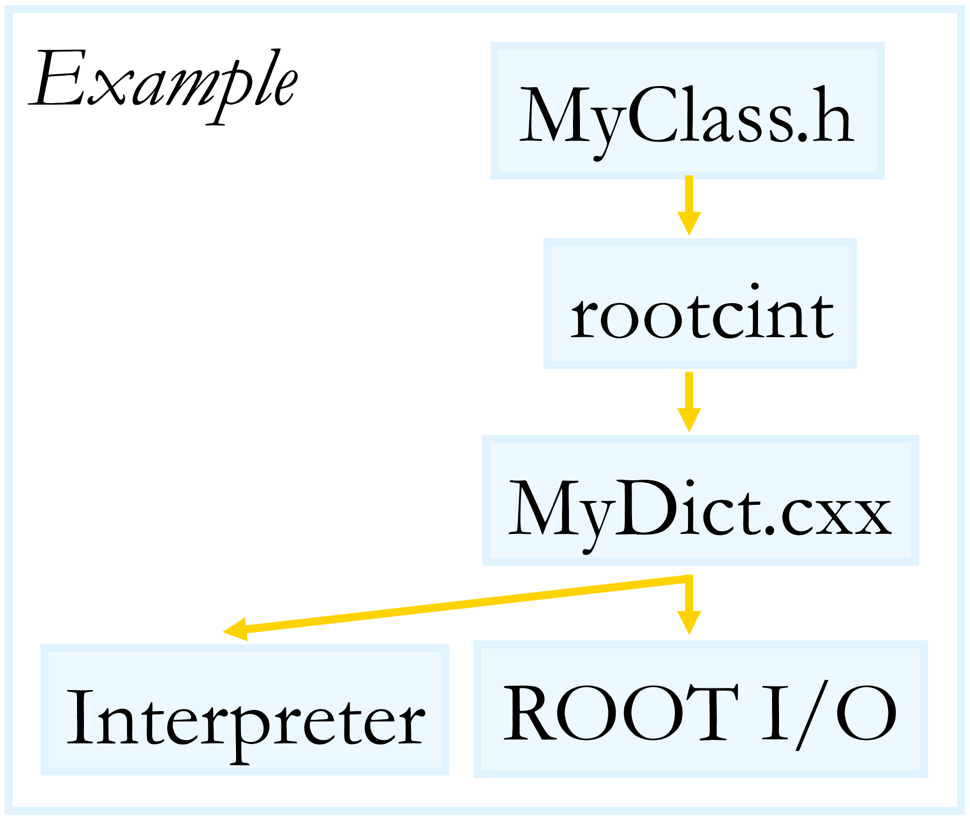
```
ClassBuilder("A").
AddFunction("f", Type("int"))
```

And how to pass arguments – using **stubs**:

```
void stub_A_f(void* args[]){
  A::f((int)args[0]); }
```

Overview Of Reflection Data





CINT And ACLiC

The plus in

```
root [0] .L MyCode.cxx+
```

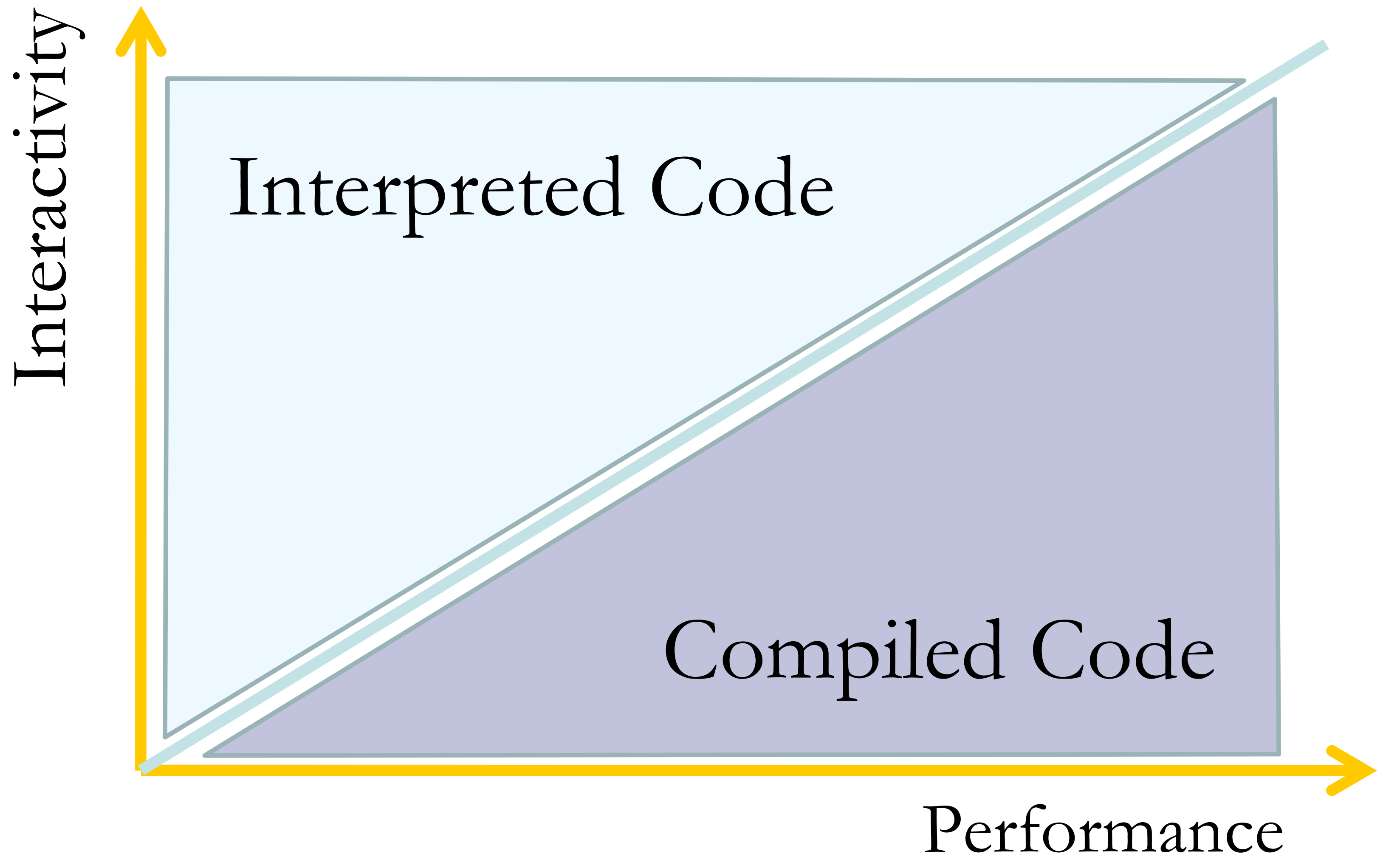
Invokes:

- dictionary generator
- compiler
- linker

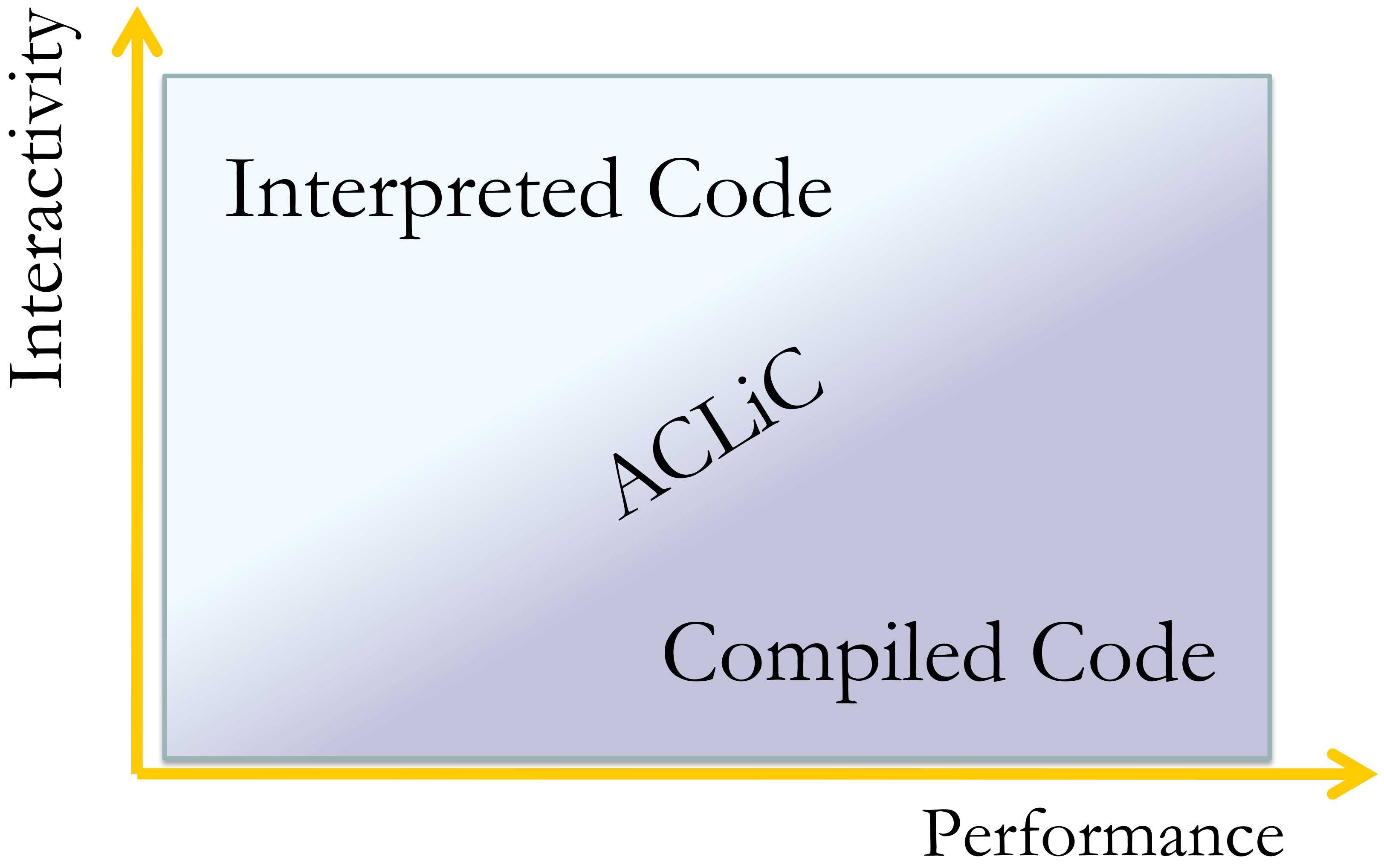
Any platform, any compiler, with any libraries!

Trivial transition from interpreted to compiled!

Traditional Decomposition



Less Walls With ACLiC



LLVM

Alternative to CINT based on LLVM Compiler Infrastructure Project. See llvm.org

LLVM is "much more than a compiler"

Modular design, allows us to hook e.g. into

- output of parser,
- language-independent code representation (IR)

Offers JIT, bytecode interpreter…

Summary: Interpreters

Wide spectrum of applications and solutions

Python and CINT are widespread and reasonable options with different use cases

Can make it easier to use higher level constructs.

Easier to share

Can not replace compiled code:

- Performance
- Difficulty in debugging and maintaining large code base
- Lack of static type checks

Summary: C++ Interpreter

Interface between interpreter and compiled code is essential but delicate.

CINT’s transparent transition between interpreted and compiled world is a huge benefit

Continually enhancing our C++ interpreter based on many years of practical experience.

Backups Slides

C++ and Data (slide 39+)

What is CINT (slide 62+)

C++ and Data

An overview of serialization in C++

ACAT 2008

Axel Naumann (CERN), Philippe Canal (Fermilab)

What Data? Why C++?

- Experiments' frameworks: C++
- Physicists' analyses: C++
- High performance, collaborative development,…
- Experiments' data: C++ objects on tape

→ Serialization with C++!

Ingredients

Storing data means:

- Reflection: types? members?
- Introspection: what is its type?
- Object instantiation from type / destruction
- I/O: memory to disk and back (endianness)

- Pointer / References ([un]swizzling)
- Schema Evolution: enabling changes

Ingredients

» Language support requested:

- Reflection: types? members?
- Introspection: what is its type?
- Object instantiation from type / destruction
- I/O: memory to disk and back (endianness)

» I/O framework's job (e.g. ROOT):

- Pointer / References ([un]swizzling)
- Schema Evolution: enabling changes

Ingredients

» Language support requested:

- Reflection: types? members?
- Introspection: what is its type?
- Object instantiation from type / destruction
- I/O: memory to disk and back (endianness)

» I/O framework's job (e.g. ROOT):

- Pointer
- Schema Evolution: enabling changes

not covered here!

Serialization Everywhere

Other languages offer some of these ingredients, usually excluding pointer / reference swizzling, schema evolution:

- Java

```
class C implements Serializable
```

- Python

```
cPickle.dump(myObj, file, -1)
```

- .NET

```
[Serializable] class C
```

Serialization In C++?

C++ supports none of these ingredients:

- Reflection: **missing**
- Introspection: basic, fragile (`typeid`)
- Object instantiation from type: **missing**
- Raw I/O: yes, endianness: **missing**
- Pointer Swizzling: **missing**
- Schema Evolution: **missing**

Serialization Not In C++

None of the relevant ingredients supported

Must rely on external packages, using e.g.

- templates (type description level)
- `typeid` (introspection)
- CPP macros

Look at consequence of matching external packages to custom code

Intrusiveness

Changes types to add serialization support

Often base, friend, etc.

» Do I need to change the header?

Example:

Microsoft's MFC: inheritance from CObject;

```
class C: public CObject
```

Reflex: no requirements

Dictionary: What's In A Type

Explicit enumeration of members / bases?

Example: boost::serialization (paraphrased)

```
class C {
  void serialize(Archive& ar) {
    ar & m;
  }
  std::string m;
};
```

Reflex: dictionaries from headers; part of build

Object Construction

Serialization:

1. create object in memory given its type name,
2. fill object with stored data

Object construction needs access to constructor

Why not add access to *all* public functions?

Interpreter Access

Reflection allows interactive use of classes

```
gROOT->ProcessLine("A::f(1)")
```

Interpreter knows type A, function f(), e.g.:

```
ClassBuilder("A").
AddFunction("f", Type("int"))
```

And how to pass arguments – using **stubs**:

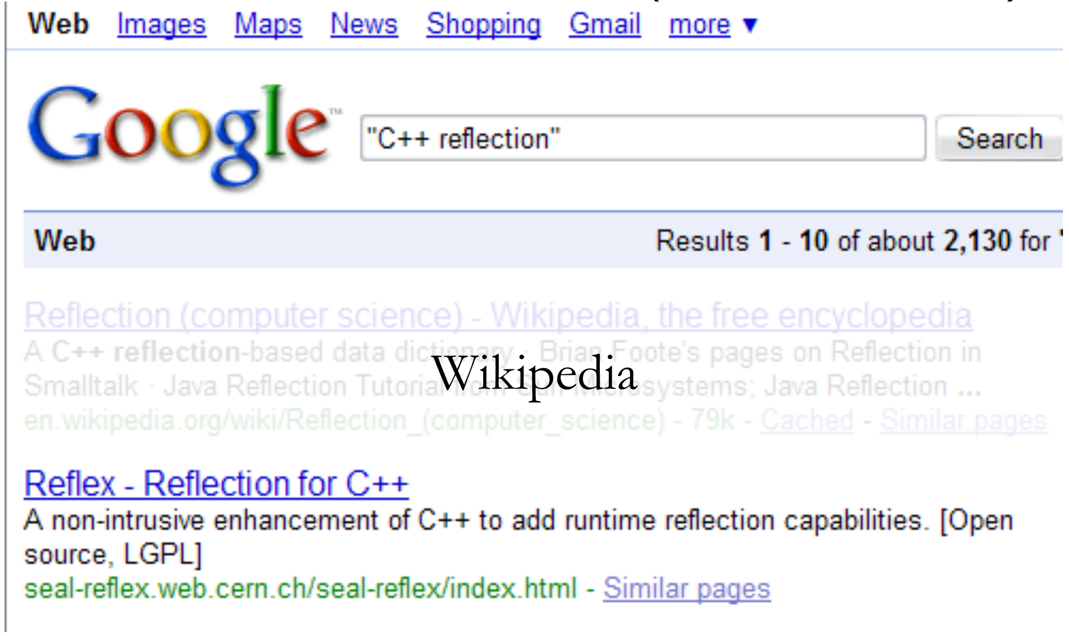
```
void stub_A_f(void* args[]){
  A::f((int)args[0]); }
```

Reflection – Market Overview

Database of available types and their structure

Main available C++ reflection libraries (unordered) :

- XCppRefl
- CppReflection



Web Images Maps News Shopping Gmail more ▼

"C++ reflection" Search

Web Results 1 - 10 of about 2,130 for

Reflection (computer science) - Wikipedia, the free encyclopedia
A C++ reflection-based data dictionary · Brian Foote's pages on Reflection in Smalltalk · Java Reflection Tutorial for ... systems; Java Reflection ...
en.wikipedia.org/wiki/Reflection_(computer_science) - 79k - Cached - Similar pages

Wikipedia

Reflex - Reflection for C++
A non-intrusive enhancement of C++ to add runtime reflection capabilities. [Open source, LGPL]
seal-reflex.web.cern.ch/seal-reflex/index.html - Similar pages

- ROOT's Reflex:
  Google's #1 product for "C++ reflection" – no wonder industry cares about it…

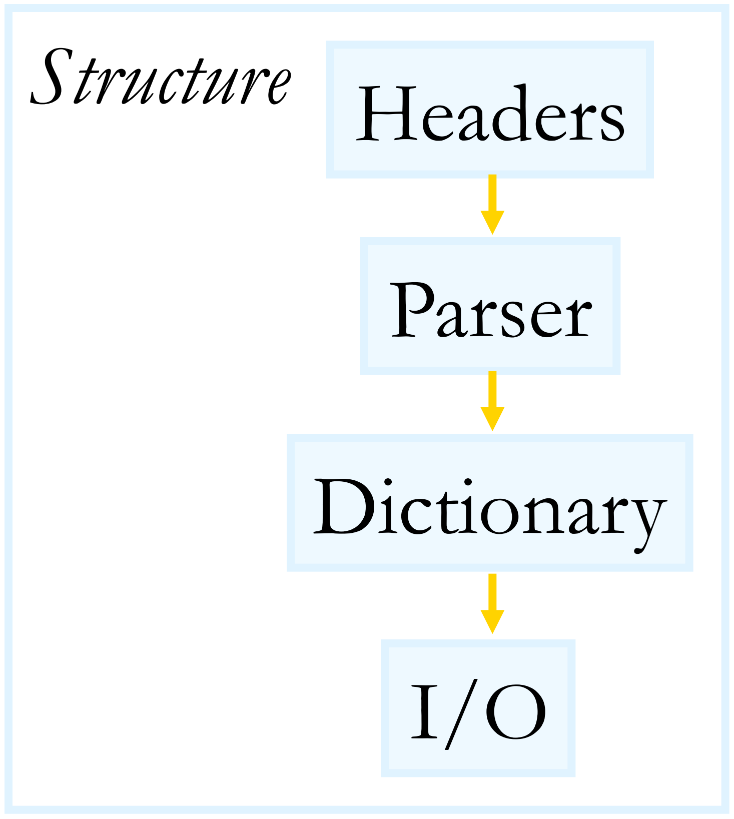
Reflection and Dictionaries

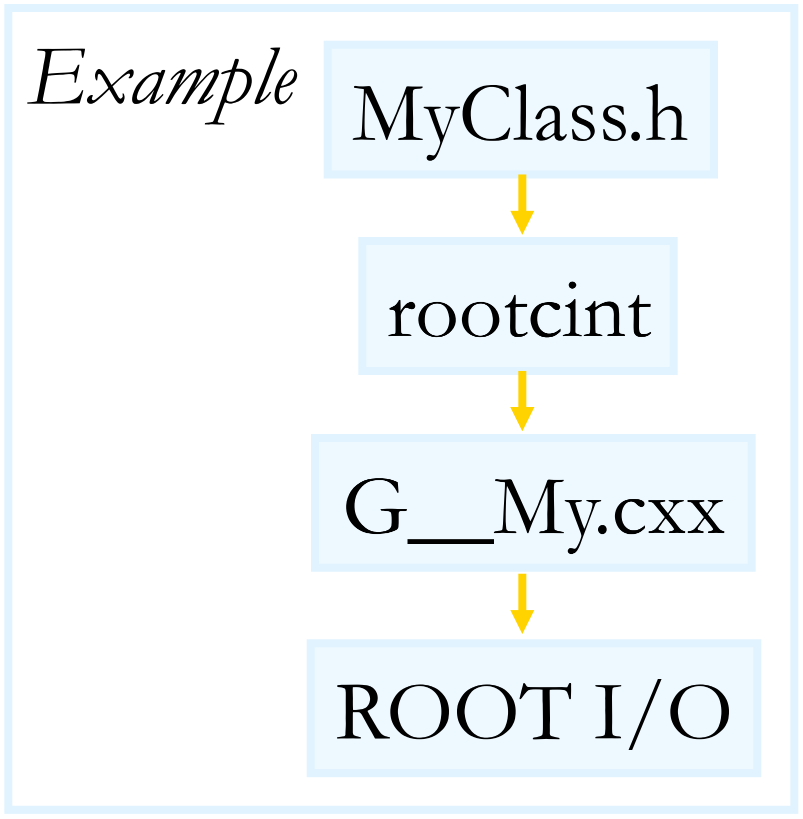
Reflection is database, dictionary is data.

Refection data generated from

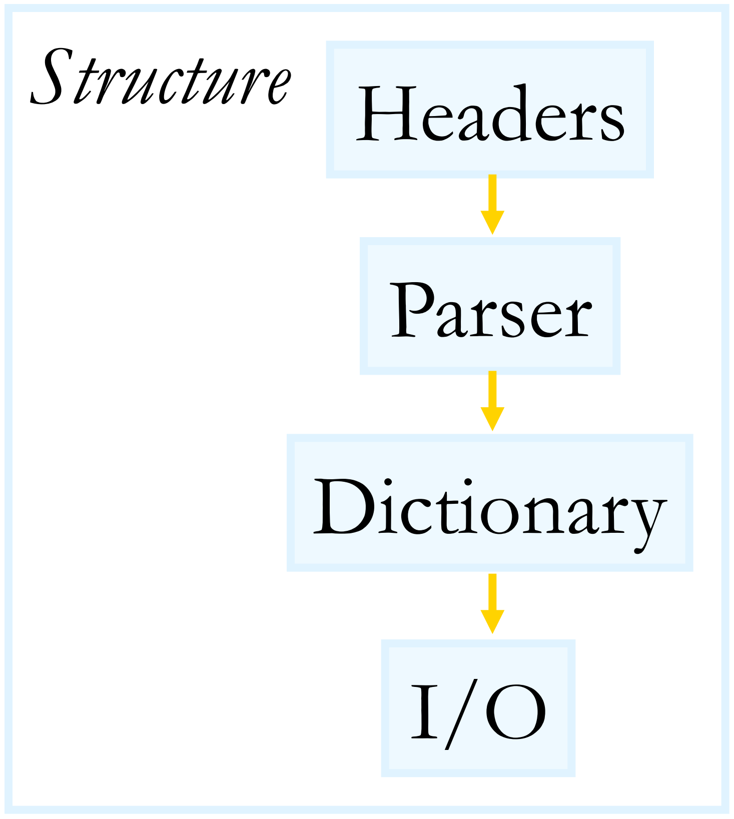
- user: `Reflect::AddClass("MyClass")`
- headers using modified compiler: GCCXML
- headers using custom parser: CINT
- debug information

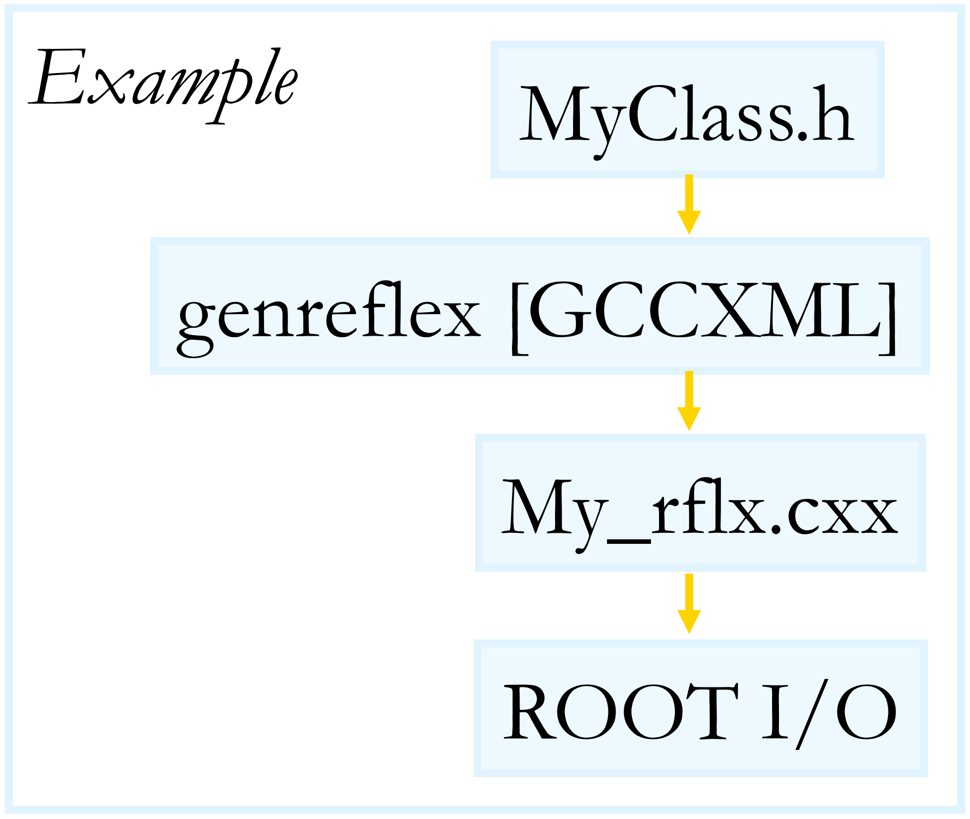
Overview Of Reflection Data





Overview Of Reflection Data





Reflection Data

Dictionary creation: time consuming

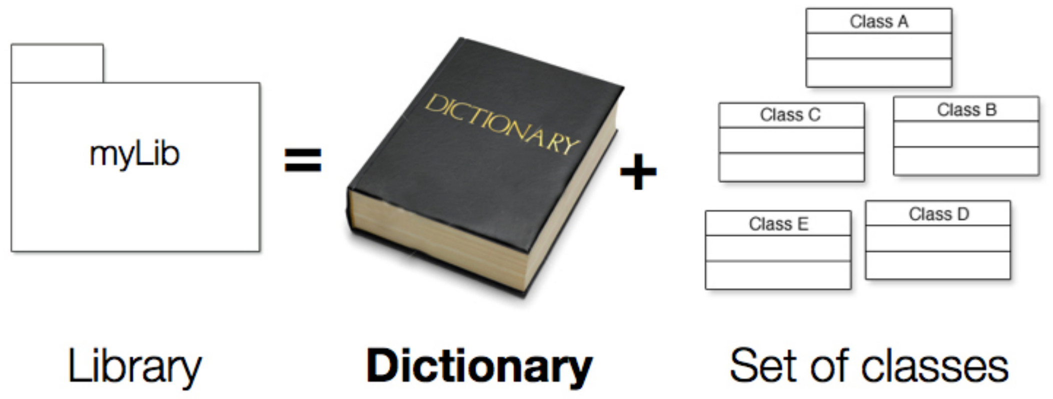
Currently persistent as generated C++, excerpt:

```
static void G__setup_memfuncTObjArray(void) {
G__tag_memfunc_setup(G__get_linked_tagnum(&G__G__ContLN_TObjArray));
G__memfunc_setup("BoundsOk",805,(G__InterfaceMethod) NULL, 103,
  -1, G__defined_typename("Bool_t"), 0, 2, 1, 2, 8,
  "C - - 10 - where i - 'Int_t' 0 - at", (char*)NULL, (void*) NULL, 0);
G__memfunc_setup("Init",404,(G__InterfaceMethod) NULL, 121, -1, -1, 0, 2, 1, 2, 0,
   "i - 'Int_t' 0 - s i - 'Int_t' 0 - lowerBound", (char*)NULL, (void*) NULL, 0);
G__memfunc_setup("TObjArray",878,G__G__Cont_81_0_5, 105,
   G__get_linked_tagnum(&G__G__ContLN_TObjArray), -1, 0, 2, 1, 1, 0,
   "i - 'Int_t' 0 'TCollection::kInitCapacity' s i - 'Int_t' 0 '0' lowerBound",
   (char*)NULL, (void*) NULL, 0);
G__memfunc_setup("TObjArray",878,G__G__Cont_81_0_6, 105,
   G__get_linked_tagnum(&G__G__ContLN_TObjArray), -1, 0, 1, 1, 1, 0,
   "u 'TObjArray' - 11 - a", (char*)NULL, (void*) NULL, 0);
G__memfunc_setup("operator=",937,G__G__Cont_81_0_7, 117,
   G__get_linked_tagnum(&G__G__ContLN_TObjArray), -1, 1, 1, 1, 1, 0,
   "u 'TObjArray' - 11 - - ", (char*)NULL, (void*) NULL, 0);
G__memfunc_setup("Clear",487,(G__InterfaceMethod) NULL,121, -1,-1, 0, 1, 1, 1,
   0, "C - 'Option_t' 10 '\"\"' option", (char*)NULL, (void*) NULL, 1);
```

Reflection Data

Dictionary sources compiled, linked into library

Become part of *enhanced* library:



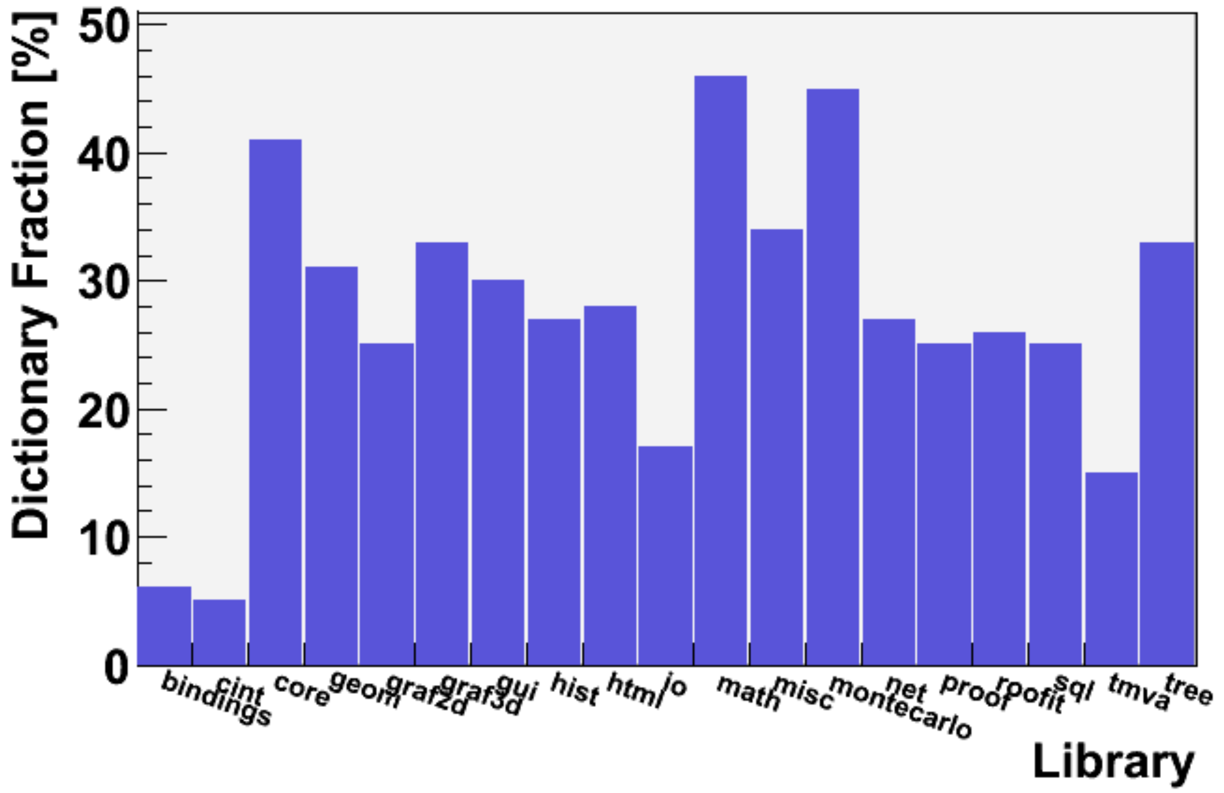
Alternative: keep separate dictionary library

Reflection Data

Dictionaries contain large amount of data

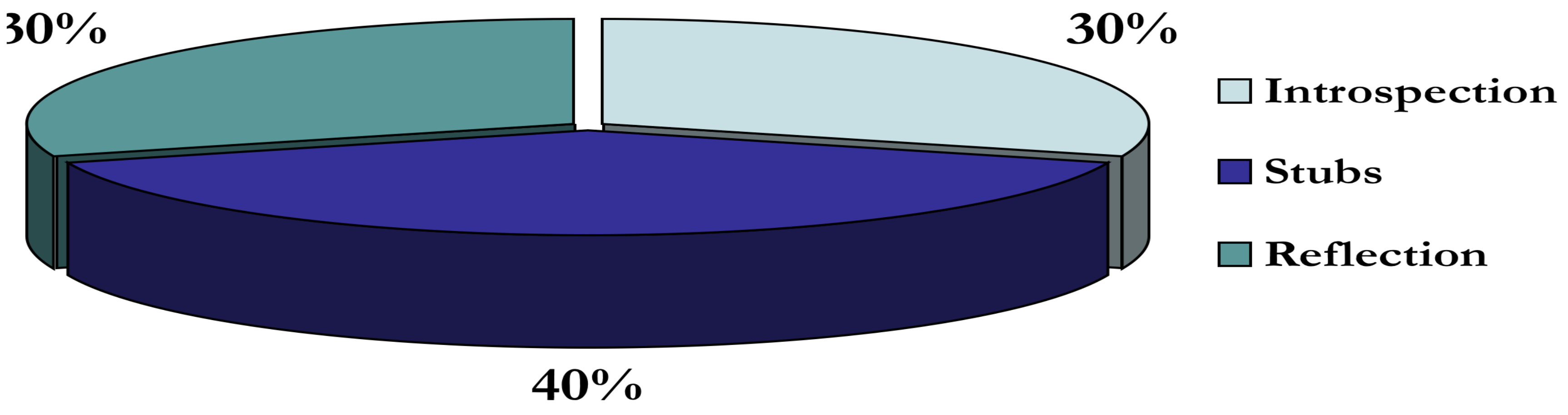
About 1/3 of library size: depends on amount of

- templates,
- functions,…



Dictionary Size

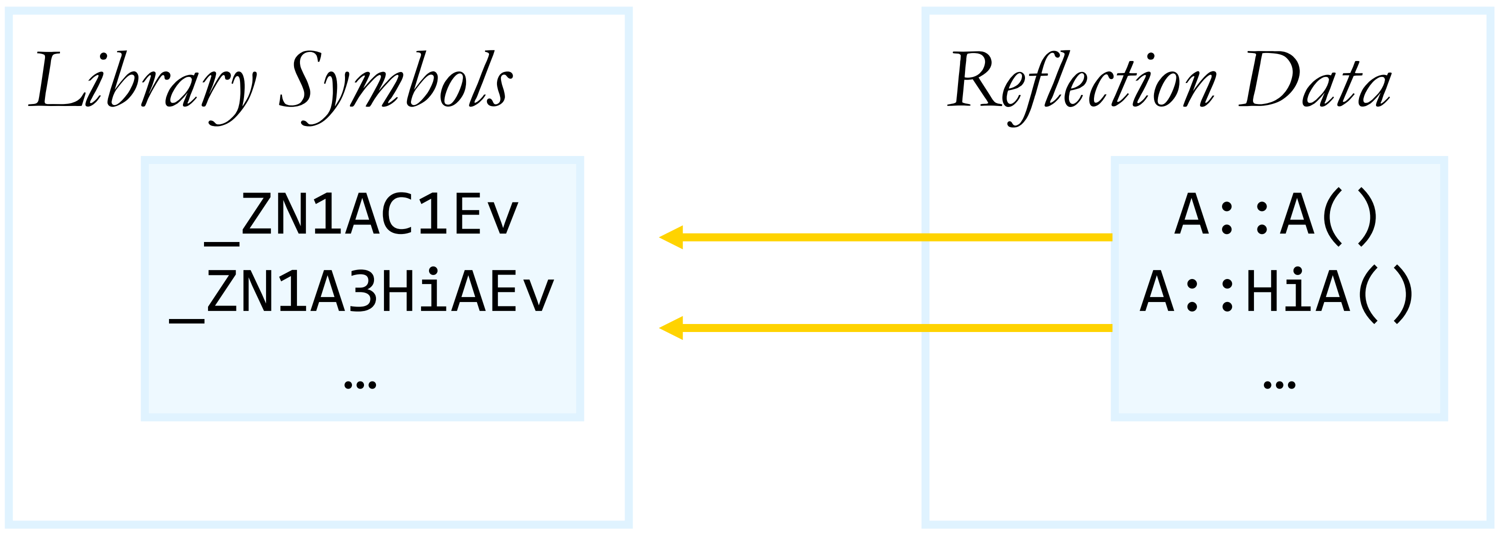
- Dictionary stub for each function
- Entry for each type
- Entry for each member (data, function)
- Names, types, parameters…



Reflection Data Optimization

Reflection data, function access optimization

- Load on demand
- Less / no copies of strings
- No stubs (use library symbols instead)

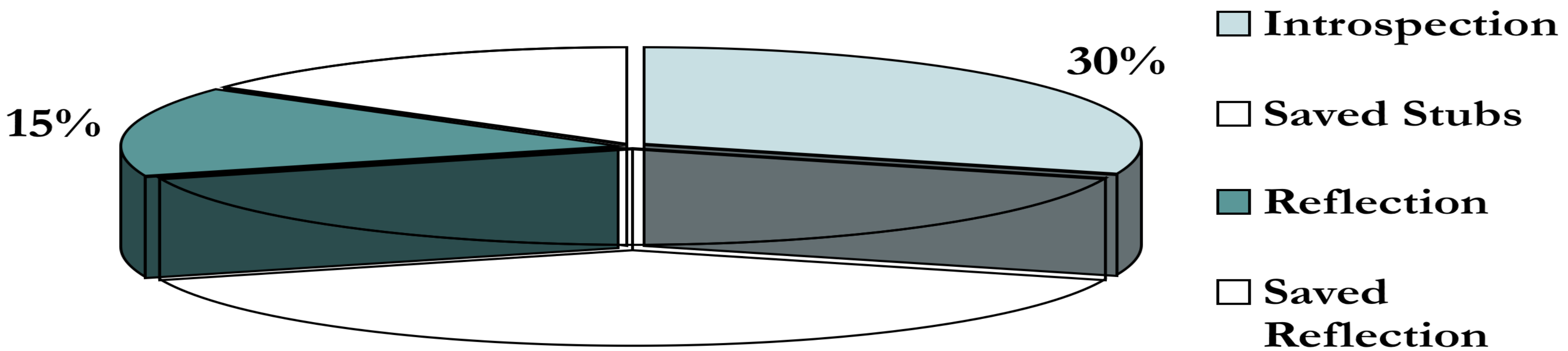


Reflection Data Optimization

ROOT will soon serialize reflection objects

Proof of concept already implemented

- Reduce disk space
- Improve build (no libraries)
- Unload when done



Summary: C++ And Data

An incredibly complex relationship

Understood, mastered, optimized in HEP

Visible outside HEP, sought-after by industry

And we did not even talk about I/O…

Status and Future of CINT

Reflex as Reflection Database
Object-Oriented CINT
Multi-Threading

Masaharu Goto, Agilent • Philippe Canal, Fermilab • Stefan Roiser, CERN
Paul Russo, Fermilab • Axel Naumann, CERN

Status and Future of CINT

What is it? Why does ROOT need it?

CINT's current status

CINT's future:

- Dictionary developments
- Object oriented design
- Multithreading support

What is CINT?

Reflection data manager

Dictionary generator

C++ Parser

Code and library manager

Interpreter

Started in 1991 by Masaharu Goto, originally in C

>300k *real* LOC (excluding comments / empty lines)

ROOT is major "customer" of CINT

What is CINT? Reflection

CINT manages reflection data (type information):

1. Which types are defined?

   Use case: THtml generates doc for all known classes

```
     root [0] THtml h
root [1] h.MakeAll(kTRUE)
            ...
  346 htmldoc/TAxis.html
345 htmldoc/TBaseClass.html
344 htmldoc/TBenchmark.html
  343 htmldoc/TBits.html
   342 htmldoc/TBox.html
 341 htmldoc/TBrowser.html
  340 htmldoc/TBtree.html
 339 htmldoc/TBuffer.html
            ...
```

What is CINT? Reflection

CINT manages reflection data (type information):

2. Which members do they have?
3. Where are they? (Member offset from object address)

Use case: I/O writes all members to file

```
root [0] TH1::Class()->GetStreamerInfo()->ls()
    StreamerInfo for class: TH1, version=5
                         ...
     Short_t          fBarOffset    offset=656
     Short_t          fBarWidth     offset=658
     Double_t         fEntries      offset=664
     Double_t         fTsumw        offset=672
     Double_t         fTsumw2       offset=680
```

What is CINT? Reflection

CINT manages reflection data (type information):

4. Which functions does TNeuron have?

   Use case: function lookup in interpreter

```
      root [0] TNeuron neuron
    root [1] neuron.MoreCoffee()
Error: Can't call TNeuron::MoreCoffee()
```

What is CINT? Reflection

CINT manages reflection data (type information):

5. Call a function
   Use case: Signal / Slot mechanism in GUI,
   e.g. sort TBrowser entries by name if name column
   header is clicked

```
Connect("Clicked()", "TRootBrowser", fBrowser,
  Form("SetSortMode(=%d)", kViewArrangeByName));
```

What is CINT? Reflection

CINT manages reflection data (type information):

1. Which types are defined?
2. Which members do they have?
3. Where are they?
4. Which functions does TNeuron have?
5. Call a function

What Is CINT?

Reflection data manager

Dictionary generator

C++ Parser

Code and library manager

Interpreter

ROOT's dictionary generator
rootcint is based on CINT

What Is CINT?

Reflection data manager

Dictionary generator

C++ Parser

Code and library manager

Interpreter

CINT remembers which macros libraries were loaded; can re-parse for template instantiations

What Is CINT?

Reflection data manager

Dictionary generator

C++ Parser

Code and library manager

Interpreter

ROOT prompt

```
.x Macro.C
gROOT->ProcessLine(...)
```

Current Status

Major developments since last workshop:

Many limitations removed, e.g. concerning array vs. scalar, auto-loading

Many new features, e.g. AMD64, MS VisualC++ 2005 support

Reduced memory footprint (10MB when running ROOT's benchmarks.C)

New build system both for CINT itself (configure) and ROOT's CINT (cintdlls-Makefile)

Bug fixes

Dictionary Size

Dictionary mainly consists of

call wrappers: translate string **`"TObject::GetName()"`** to function call

Function calls to setup dictionary: add **`"TObject"`**, add its function **`"GetName()"`** etc

Public re-definition of classes to inspect their (otherwise private) members

Dictionary Size

Dictionary mainly consists of

call w

to

Extract address of "TObject::GetName()" from library, forward calls directly too that address

Function calls to setup dictionary: add **`"TObject"`**, add its function **`"GetName()"`** etc

Public re-definition of classes to inspect their (otherwise private) members

Dictionary Size

Dictionary mainly consists of

call w

to

Extract address of "TObject::GetName()" from library, forward calls directly too that address

Func

add

Store dictionary data in precompiled header file, instead of compiled dictionary

Public re-definition of classes to inspect their (otherwise private) members

1. finish CINT7 (Reflex)

3. on-the-fly dictionaries (template dicts)
4. object-oriented CINT (class G__Interpreter)
5. multi-threading support
6. byte-code compiler (loop, scoping problems

Dictionary Size

Dictionary mainly consists of

call w

to

Extract address of "TObject::GetName()" from library, forward calls directly too that address

Func

add

Store dictionary data in precompiled header file, instead of compiled dictionary

Publi

(ot

Calculate member inspection data on the fly or examine (compiler dependent) memory layout

On-Demand Dictionary

Currently: dictionaries for *all* types

On-demand: generate and cache *needed* dictionaries

Need class **`"MyClass<int>"`**, but no dictionary

1. parse MyClass's header
2. create dictionary for
3. compile (ACLiC) / load dictionary

Great for templates: no 100 dicts for 100 template specializations "just in case"

Summary

CINT amazingly stable: very few lines changed, virtually no API changes; well maintained

Shortcomings known

Remedy: alternative to CINT based on LLVM Compiler Infrastructure Project.