# ROOT and C++11

## ROOT Users Workshop 2013

Benjamin Bannier

Stony Brook University

March 13, 2013

# About me and Disclaimers

Hi, I am Benjamin Bannier.

- graduate student at Stony Brook University
- Experimental Heavy Ion Physics with PHENIX/RHIC

I first used ROOT around 2004, and before that my only "programming experience" was "data processing" with `awk`.

I *do like* C++ and use it daily.

But for plotting I use ROOT with Python or matplotlib.

I have to realize daily that many things are too complex for me.

# Why on earth C++?

## Efficiency

## Type system

- statically typed
- reasonably extensible type system
- high-level abstractions, even with zero extra runtime cost

## Multiple paradigms

- interoperation of different programming styles

## C compatibility

- support for the C machine model
- reasonably easy to interact with C APIs

# But . . . !

## C++ is a *huge language*.

- C and C++ standard library
- C, C++, template metaprogramming, preprocessor macros

*You like C++ because you're only using 20% of it. And that's fine, everyone only uses 20% of C++, the problem is that everyone uses a different 20% :)* – kingkilr, in /r/programming

## C++ can be extremely unsafe.

*C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.*

– B. Stroustrup, *Did you really say that?*

The intersection of {*valid C++ programs*} and {*code you want to read/write/maintain*} is tiny.

# C++ evolution

C++ stays mostly backwards-compatible with C.

When looking at the evolution of C++ what I find *most interesting* is what is *added*

- higher level abstractions
- tools which allow writing safer code

In C++ design decisions are more and more abstracted into types, e.g.

- a *reference* is essentially a *"pointer which cannot be 0"*
- consistency with constructors and destructors (e.g. with RAII)
- `const` to annotate what should not change

Type constraints are documention and provide extra safety with zero cost at runtime.

## C++98, C++03

- started as a set of extensions on top of C
- 1st edition of *The C++ Programming Language* in 1985
- standardized in 1998, bug fix in 2003 (nothing new)
- compilers with support for all features relatively late
- this is all there is to C++ in ROOT

## C++98, C++03

- started as a set of extensions on top of C
- 1st edition of *The C++ Programming Language* in 1985
- standardized in 1998, bug fix in 2003 (nothing new)
- compilers with support for all features relatively late
- this is all there is to C++ in ROOT

## C++ Technical Report 1 (TR1)

- no new standard, only possible additions to standard library (often from Boost); draft in 2005, published 2007
- `tr1::shared_ptr`, `tr1::weak_ptr`, reference wrappers
- `<type_traits>`
- `tr1::function`, `tr1::bind`, `tr1::mem_fun`
- `tr1::tuple`, `tr1::array`, and unordered sets and maps
- `<random>`, additional mathematical functions
- largely supported by most compilers (or via Boost.TR1)

## C++11 (selected)

- in the works for a long time, published 2011
- N3337 is a free draft very close to the published standard

- most of TR1
- variadic templates
- uniform initialization
- $\lambda$ functions
- concurrency support
- rvalue references
- type deduction
- range-based for loops
- user-defined literals
- ...

C++11 brought the language up-to-date with existing best practices:

- leverage the type system: *compile-time errors are better than runtime errors*
- safer, high-level abstractions with zero extra runtime cost
- more compact: *code not written cannot introduce bugs*
- as performant as low-level code (or better)

C++11 allows to write beginner-friendly, safe and efficient code.

# Example applications

# Anonymous $\lambda$ functions and function objects

## C++98

```
1  // function pointers
2  int sum2(int x, int y) { return x+y; }
3  int (*sum2p)(int, int) = sum2;
4  sum2p(1, 2); // int(3)
5
6  // functor
7  struct {
8    int operator()(int x, int y) { return x+y; }
9  } sum1;
10 sum1(1, 2);  // int(3)
```

- need to refer to something defined non-locally
- hard to store and pass polymorphically

## C++11

```
1  auto sum1 = [](int x, int y) { return x+y; };
2  int (*sum1p)(int, int) = sum1;
3
4  function<int(int, int)> sum2 = sum1;
5  int (*sum2p)(int, int) = sum2; // doesn't work
6
7  // defined somewhere: int f(int, int)
8  function<   int(   int,    int)> fi = f;
9  function<double(double, double)> fd = f;
```

- `function` and function pointers live in disconnected worlds.
- polymorphic $\lambda$s might be coming soon

## Support

`function` was `tr1::function`

**$\lambda$ functions** gcc-4.5, clang-3.1, icc-11.0, msvc-10.0

## What already works

```
// TF1 f("f", "x"); // formula only checked at runtime
TF1 f("f", [](double* xs, double* ps) { return xs[0]; });

TH1 *m_meas;  // measured distribution
TH1 *m0, *m1; // simulated distributions (say stat. err = 0)
TF1 sum("sum", [m0, m1](double* ms, double* ps) {
    double m = ms[0];
    double y0 = m0->Interpolate(m);
    double y1 = m1->Interpolate(m);
    return y0*ps[0] + y1*ps[1];
    }, 0, 10, 2);
m_meas->Fit(&sum); // cannot use temporary here?
```

With $\lambda$ functions much of `TFormula` is dangerous convenience functionality.

Anywhere a function pointer is used now one could use `function` objects to constrain types.

## A step further

```
1  TNtuple t("t", "", "x:y");
2
3  // t.Draw("x", "x>0");
4
5  // TTree
6  TTreeReader tr(&t);
7  TTreeReaderValue<float> x(tr, "x");
8  TTreeReaderValue<float> y(tr, "y");
9  t.Draw([&x,&y]() { return {*x, *y} ); },
10        [&x]   () { return *x > 0; });
```

# Smart pointers

Ownership issues are one of the big source of confusion and bugs in code using the ROOT API.

```
1  // Who should manage the lifetime of a return value?
2  Object* get(const string& name);
3
4  // what should happen to pointer members in class on copy?
5
6  // What are an object's dependencies?
7  TFile f("f.root", "recreate");
8  TH1* h = new TH1D("h", "", 100,0,1);
9  f.Close();
10 h->GetEntries();    // segfaults
```

Raw pointers do not help in answering any of these questions.

Smart pointers encode design decisions into types, and have largely replaced raw pointers in the C++ world.

## From TR1

`shared_ptr` reference-counted pointer, shared ownership

`weak_ptr` non-owning reference to a `shared_ptr`

## C++11

`unique_ptr` unique ownership

Designing an API with shared ownership is hard because designing consistent dependencies is hard.

Smart pointers document and abstract the design away in types.

A compiler can enfore correct use.

# One more look at example (3)

```
1  TFile f("f.root", "recreate");
2  TH1* h = new TH1D("h", "", 100,0,1);
3  f.Close();
4  h->GetEntries();    // segfaults
```

Who owns what here?

## One more look at example (3)

```
1  TFile f("f.root", "recreate");
2  TH1* h = new TH1D("h", "", 100,0,1);
3  f.Close();
4  h->GetEntries();    // segfaults
```

Who owns what here?

```
1  TList* TDirectory::fList;    // f holds pointer to h
2  TDirectory* TH1::fDirectory; // h holds pointer to f
```

We need to specify who should outlive the other.

Here h depends on f, but not the other way around.

*Just one idea*:

```
1  shared_ptr<TDirectory> gDirectory;
2
3  // only allow TFile to be created with factory
4  shared_ptr<TFile> TFile::Open(..) {
5    // ...  TFileOpenHandle *fh = 0;
6    return std::shared_ptr<TFile>(fh->GetFile());
7  }
8
9  // h could have shared ownership of f
10 shared_ptr<TDirectory> TH1::fDirectory
```

Here h depends on f, but not the other way around.

*Just one idea*:

```
shared_ptr<TDirectory> gDirectory;

// only allow TFile to be created with factory
shared_ptr<TFile> TFile::Open(..) {
  // ...  TFileOpenHandle *fh = 0;
  return std::shared_ptr<TFile>(fh->GetFile());
}

// h could have shared ownership of f
shared_ptr<TDirectory> TH1::fDirectory
```

I think this is a part where the ROOT API is underspecified.

Annotating pointer use policy would be a start so one could selectively activate smart pointers in completed sections.

# Concurrency

## C++98

- no notion of concurrency in standard library
- pthreads is a widely available multithreading C API
  - little type safety
  - doesn't integrate too nicely into C++ code
- developing concurrent code is for experts

## C++11

- high-level abstractions for
  - asynchronous code
  - multithreaded code
  - lock management
  - ...
- even beginners might use it

```
TH1* h = ... ;
vector<TF1> fs = { ... };

vector<future<TFitResultPtr>> fits;

for (auto& f : fs)
  fits.push_back( async([&]() {

        return h->Fit(&f, "S");
        }) );

for (auto& fit : fits)
  fit.get().Get()->Print();
```

```
TH1* h = ... ;
vector<TF1> fs = { ... };

vector<future<TFitResultPtr>> fits;

for (auto& f : fs)
  fits.push_back( async([&]() {

        return h->Fit(&f, "S");
        }) );

for (auto& fit : fits)
  fit.get().Get()->Print();
```

`TH1::Fit` not `const` — one shouldn't expect it to be thread-safe!

```
TH1* h = ... ;
vector<TF1> fs = { ... };

vector<future<TFitResultPtr>> fits;
mutex h_m;
for (auto& f : fs)
  fits.push_back( async([&]() {
        lock_guard<mutex> lock(h_m); // yes, boring now
        return h->Fit(&f);
        }) );

for (auto& fit : fits)
  fit.get().Get()->Print();
```

```
TH1* h = ... ;
vector<TF1> fs = { ... };

vector<future<TFitResultPtr>> fits;
mutex h_m;
for (auto& f : fs)
  fits.push_back( async([&]() {
        lock_guard<mutex> lock(h_m); // yes, boring now
        return h->Fit(&f);
        }) );

for (auto& fit : fits)
  fit.get().Get()->Print();
```

- `TFitResultPtr::Get` and `TF1::Print` are `const` – would expect this to work
- might still break if object in `future` doesn't stay consistent

## Library writer guidelines for thread-safe code

- **always** make sure objects stay consistent
- `const` methods are always thread-safe
  - use `mutable` members with internal synchronisation (e.g. by locking) if needed
- non-`const` methods might need synchronisation by user

## Optimizations (for performance or maintenance)

- avoid locking: minimize shared or global state
- no seriously

**Making a library thread-safe leads to a cleaner design.**


BUT WAIT!
Pure
Baking Soda
THERE'S MORE!

# Uniform initialization, initializer lists

*Consistent object construction*

```
TH1D h{"h", "", 100,0,10};
int a[]{1, 2, 3};
std::vector<int> v{1, 2, 3}
```

Makes it so easy to create objects that they can be used much more widely. e.g. to create temporaries

```
int f(const array<int, 3>& a) { return a[1]; }
f({1, 2, 3}); // int(2)

// why not e.g.
TH3::TH3(const char* name, const char* title,
         TAxis ax1, TAxis ax2, TAxis ax3);
TH3D h3("h3", "", {100, 0, 10}, {10, 0, 10}, {300, 0, 20});

// (even better with e.g. TAxis::TAxis(vector<double>))
```

# Range-based `for` loops

*Sugar-coated iteration*

```
1  ListT list;
2  for (auto& element : list) { /* do something with element */ }
```

The interface `ListT` needs to provide

```
1  for (auto __begin = begin-expr, // begin(T) or T::begin()
2            __end = end-expr;   // end(T)   or T::end()
3       __begin != __end;          // iterator operator!=
4       ++__begin ) {              // iterator operator++
5    for-range-declaration = *__begin;
6    // statement
7  }
```

```
1  // only non-const version here
2  struct TIteratorPtr {
3    TIterator* it = nullptr;
4    TIteratorPtr(TIterator* it) : it(it) { }
5
6    TIteratorPtr& operator++() {
7      if (not it->Next()) it = nullptr;
8      return *this;
9    }
10   TObject* operator*() { return **it; }
11   bool operator!=(const TIteratorPtr& rhs) const {
12     return it != rhs.it;
13   }
14 };
15
16 TIteratorPtr begin(TCollection& c) {
17   return ++TIteratorPtr(c.MakeIterator());
18 }
19 TIteratorPtr end  (TCollection& c) { return nullptr; }
```

# std::tuple from TR1

*A container for heterogeneous data*

```
1  tuple<double, int> tup(1.0, 42);
2  // auto tup = make_tuple(1.0, 42);
3
4  auto& v1 = get<0>(tup);          // double& v1
5  get<0>(tup) = 2.2;               // now tup = (2.2, 42)
6
7  tuple<int, int>    tup2 = tup; // tup2 = (2, 42)
8  tuple<string, int> tup3 = tup; // won't compile
```

This could allow a safe way to pass arguments to `TTree::Fill`:

```
1  template <typename ...Ts>
2  Int_t TTree::Fill(const std::tuple<Ts...>& data);
```

# Variadic templates: type-safe variadic functions

**Support** gcc-4.3/gcc-4.4, clang-2.9, icc-12.1, msvc-11.0

```
1  // TNtuple.h
2  virtual Int_t Fill(Float_t x0, Float_t x1=0, Float_t x2=0,
3                     Float_t x3=0, /* snip */
4                     Float_t x14=0);    // (1)
5  virtual Int_t Fill(const Float_t *x); // (2)
```

(1) could be generalized:

```
1  template <typename ...Ts>
2  virtual Int_t Fill(Ts... xs) {
3    array<Float_t, sizeof...(Ts)> data{{xs...}};
4    Fill(&data[0]);
5  }
```

Due to type `array<Float_t, ...>` all `xs` are checked to be `Float_t` at compile time.

Make (2) accept an `array` or `vector`?

# Summary

- C++, compilers and standard library implementations have come a long way since ROOT was started
- type constraints are an extremely useful tool to catch usage errors early
- compared to the 1990s C++ today can be much safer to use

- C++11 adds a number of abstractions and tools to make C++ code more expressive

- the disconnect between the mainstream "C++ way" and the "ROOT way" will feel more painful once C++11 features are used more
- value semantics allow to leverage language support for automatically generating code
- cling is a great opportunity to provide at least for user-visible parts a more mainstream C++ API