“Improve our Software”

Suggestions for Improvements

Properties: Reminder

Document intent of code, unify code, allows for future extensions

Example: Histogram Property in a GaudiHistoAlg

```
header:     Gaudi::Histo1DDef m_histodef
            AIDA::IHistogram1D *m_hist;
c’tor:      declareProperty( m_histoDef
                         = Gaudi::Histo1DDef(“PV3D", -0.5,10.5,11 ) )
initialize:  m_hist = book( m_histoDef )
```

can eg. add non-uniform binning *without changing C++ ‘client’ code* — just augment Gaudi::Histo1DDef and book — and to use it, just update the python configuration code.

This is an example of ‘open for extension, closed for modification’ (Meyer open/close principle)

Properties: Suggestion

Would be even nicer as:

```
header:     Gaudi::Histogram1D m_histo;

c’tor:      declareProperty( m_histo = { “PV3D”,  -0.5, 10.5, 11 } );

initialize: m_histo.book( this );

execute:    if (m_histo) m_histo.fill( … );
```

as there is only ‘one’ object — m_histo — instead of two…

This would not require a callback to respond to updated properties (if implemented right)!

and yes, I know about ‘do not book, use plot which books on demand’

— discouraged in Hlt, as it introduces overhead on filling (must check if booked on every fill)

Properties: Suggestions

possible new dedicated property types:

- filenames
  - deal with environment variables, define search paths
  - make all I/O go through IVFSSvc… (which would allow eg. ‘relocation’ of files into a zip archive — albeit due to checkpointing this has become less critical)
- tools
  - could avoid having to use ‘addTool’ in python!
  - move the ‘PUBLIC’ and ‘PRIVATE’ into a property of the property instead of ‘encoding’ it in the name
  - avoid bare tool pointers????
- TES locations
  - could add eg. alternate locations — no more RawEventLocations vs. RawEventLocation,
  - differentiate read vs. write;
  - allows (at least, makes a lot easier) static analysis during python configuration: verify ‘put’ (in one algo) precedes ‘get’ (in another algo)
- …

Properties: *Suggestions*

Make better use of existing capabilities of property parsing.

Example: L0DUConfig

(note: certainly not the only case, but it happens to be one I know well)

- replace eg. the following ‘options’ snippet:

```
ToolSvc.L0DUConfig.TCK_0x0038.Conditions = {
        { "name=[Electron(Et)>12]","data=[Electron(Et)]",“comparator=[>]",“threshold=[12]"},
        { "name=[Electron(Et)>50]","data=[Electron(Et)]","comparator=[>]", "threshold=[50]"},
}
```

- with:

```
ToolSvc.L0DUConfig.TCK_0x0038.Conditions = [
        { “name” : “Electron(Et)>12”, “data” :“Electron(Et)” ,“comparator” : “>” ,“threshold” : “12”   },
        { “name” : “Electron(Et)>50”, “data” :“Electron(Et)” ,“comparator” : “>” ,“threshold” : “50”   },
]
```

- Conditions (IMHO) should be a vector<map<string, string>>, not a vector<vector<string>>
- Leverage the power of the built-in parsing of properties, don’t do it yourself!
- And it would make manipulating this in python easier ;-)

C++11

Less “boiler plate” code

- auto
- range based loops

Lambda functions

- many uses: eg move (some) control logic out of loops
- Could always do this by defining a small struct, but not in ‘local scope’

Move semantics and RHS references

- allows for ‘perfect forwarding’ and ‘emplacement’

variadic templates

tuples

nullptr

treads, async, future

…. many, many more features….

Please, please, please take a look at:

GoingNative 2012 presentations, GoingNative 2013 presentations

```
std::vector<Hit*> hits = … ;
for (std::vector<Hit*>::const_iterator ihit = hits.begin();
     ihit!=hits.end(); ++ihit) {
        if ( !useXOnly || ( (*ihit)->layer()!=0 &&(*ihit)->layer()!=3 ) )
                continue;
        // use *ihit
        ….
}
```

```
std::vector<Hit*> hits = … ;
for (auto hit : hits ) {
        if ( !useXOnly || ( hit->layer()!=0 && hit->layer()!=3 ) )
                continue;
        // use hit
        ….
}
```

```
std::vector<Hit*> hits = … ;
auto xOnly = [](const Hit& h) { return h.layer()==0||h.layer()==3; };
auto all       = [](const Hit& h) { return true; }
auto predicate   = useXOnly ? xOnly : all ;

for (auto hit : hits ) {
        if (!predicate(*hit)) continue;
        // use hit
        ….
}
```

C++11: refactoring code

How to take advantage of C++11 ?

Need to do lots of 'tedious' changes

They can be automated with clang-modernize!

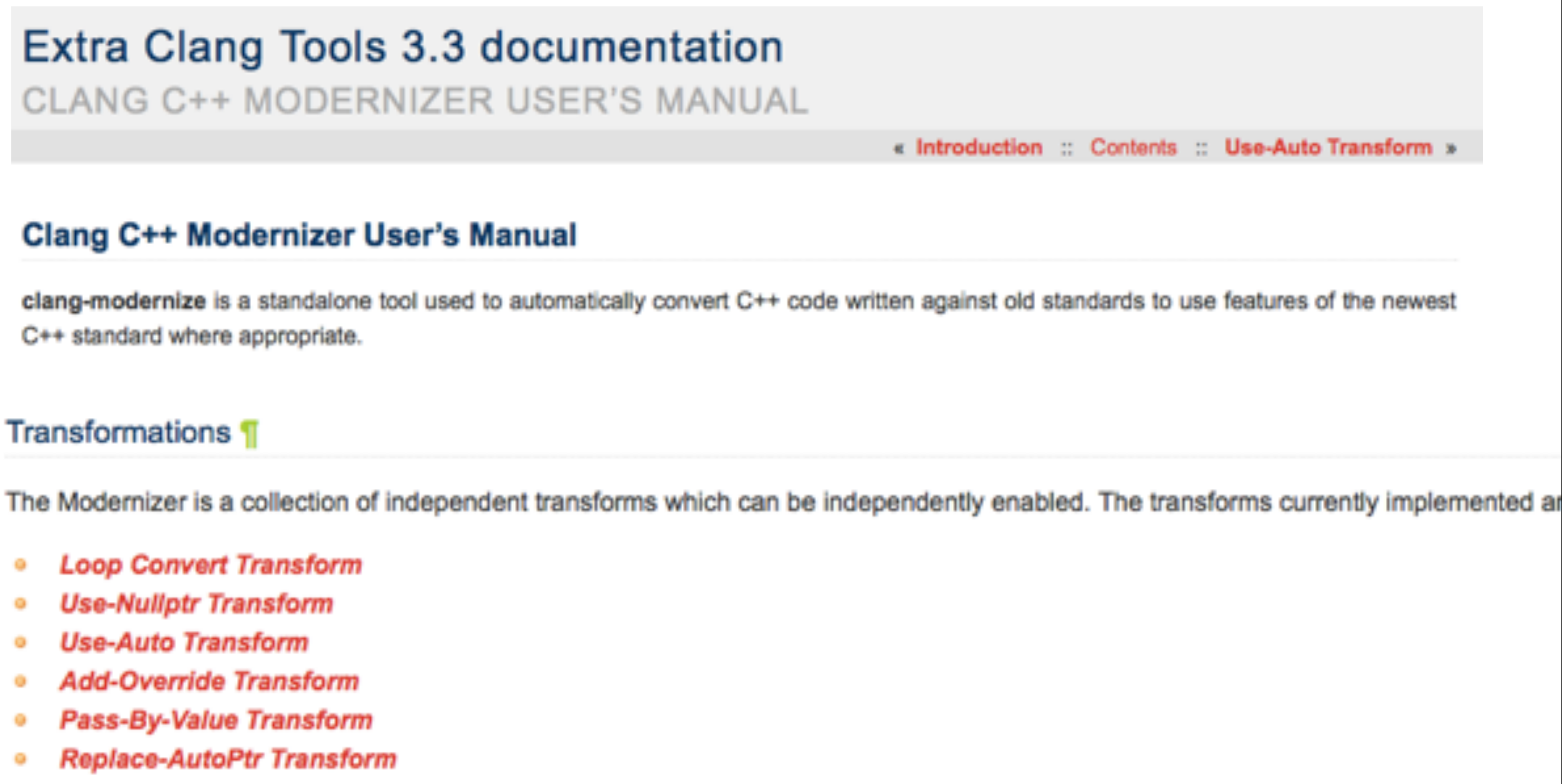
(Note: more uniform code layout can be done with clang-format)

Please, please, please take a look at:

, GoingNative 2013: The Care and Feeding of C++'s Dragons

+ get 'modern', 'better' code

- backporting more work



Extra Clang Tools 3.3 documentation

CLANG C++ MODERNIZER USER'S MANUAL

Clang C++ Modernizer User's Manual

clang-modernize is a standalone tool used to automatically convert C++ code written against old standards to use features of the newest C++ standard where appropriate.

Transformations ¶

The Modernizer is a collection of independent transforms which can be independently enabled. The transforms currently implemented ar

- *Loop Convert Transform*
- *Use-Nullptr Transform*
- *Use-Auto Transform*
- *Add-Override Transform*
- *Pass-By-Value Transform*
- *Replace-AutoPtr Transform*

Event Model: Predicates & LoKi

```
std::vector<Hit> hits = … ;
auto pred    = useXOnly ? Predicates::Hit::XOnly
                        :  Predicates::Hit::True ;

for (const auto& hit : hits ) {
        if (!pred(hit)) continue;
        // use hit
        ….
}
```

LoKi provides flexible, ‘open for extension’ framework for selection of event model objects

- (ADMASS('KS0')<35*MeV) & (VFASPF(VCHI2PDOF)<30) & (BPVLTIME('PropertimeFitter/properTime:PUBLIC') > 2.0*ps)

But cannot currently be used in REC

- Dependencies, dependencies…
- Deals with Particles, List of Particles, Tracks, Vertices, …
  - but not all event model classes

The functor/predicate definitions don’t live ‘next’ to the event model classes, but in separate LoKi packages — no guarantee that a given ‘getter’ has a matching functor/predicate.

Proposal:

1) integrate functor/predicate functionality into LHCb event model classes

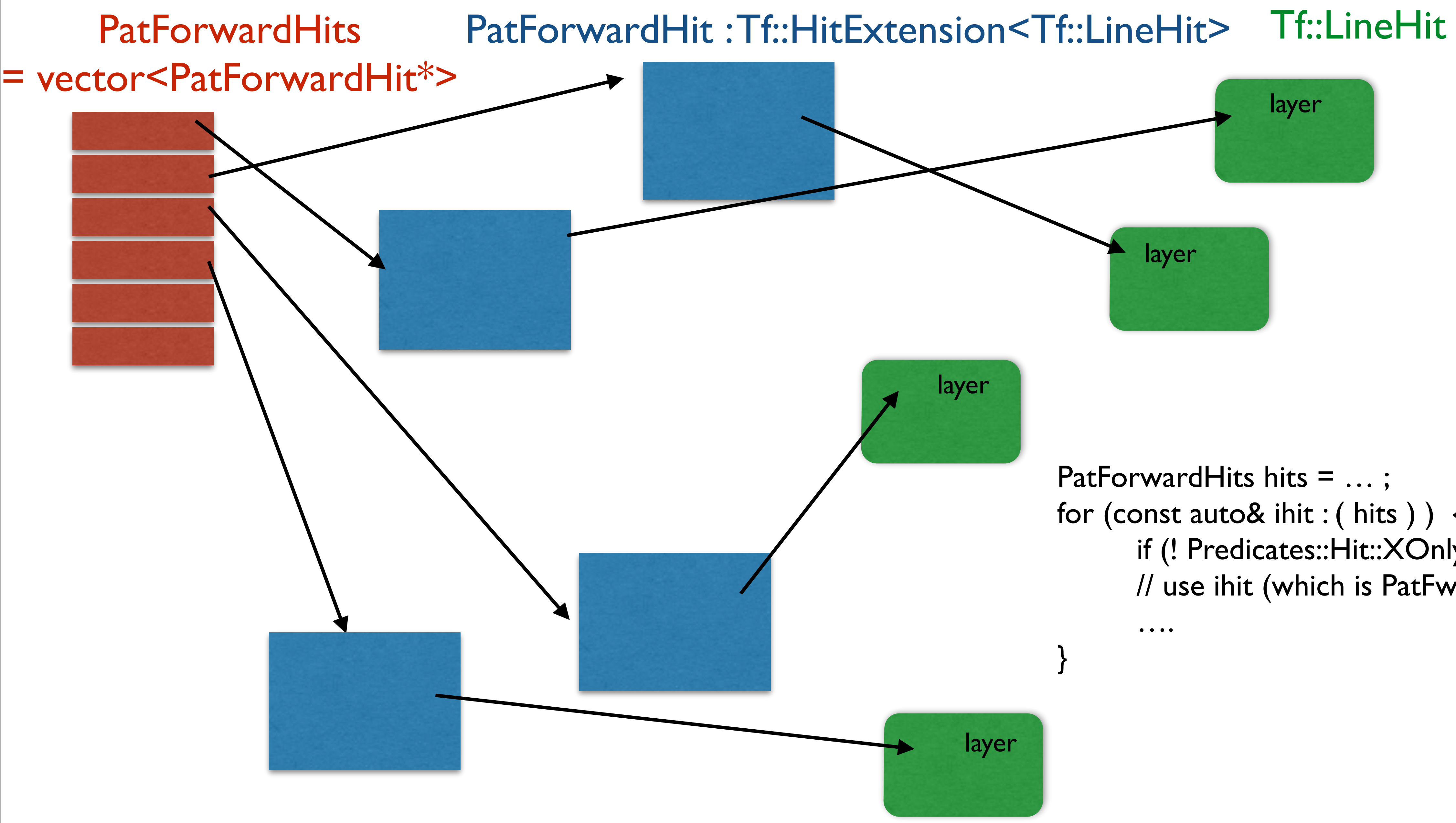
2) use GOD to generate the simple ‘getter’ based functors & predicates

3) Provide generic ‘compositing’ functionality.

4) Re-use this functionality in LoKi

Note: LoKi contains many more complicated functors/predicates — let’s take one step at a time

Data Model: Locality of Reference



PatForwardHits
= vector<PatForwardHit*>

PatForwardHit : Tf::HitExtension<Tf::LineHit>

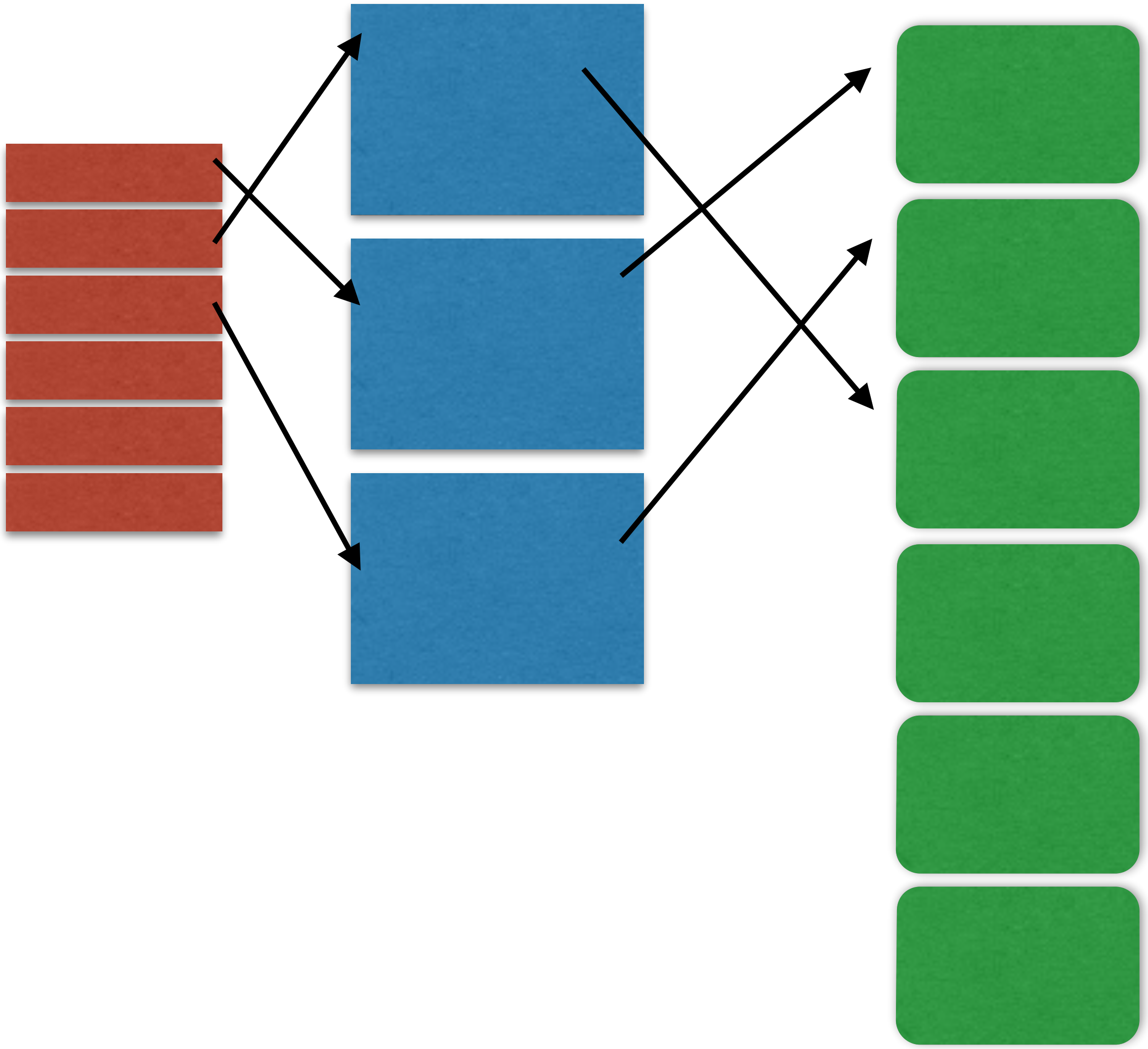
Tf::LineHit

```
PatForwardHits hits = … ;
for (const auto& ihit : ( hits ) )  {
        if (! Predicates::Hit::XOnly( ihit ) ) continue;
        // use ihit (which is PatFwdHit* )
        ….
}
```

All ‘objects’ are new’ed individually
(although MemPoolAlloc will do its best to keep them together)

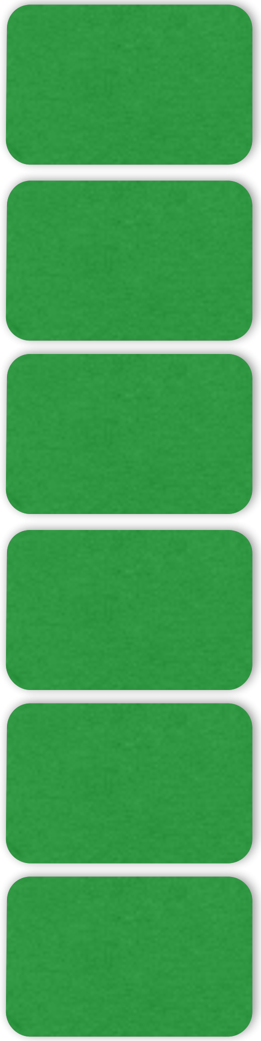
loop body may eg. use hit->layer()
This layout is not very ‘cache friendly’….

Data Model: *Collections* of Objects



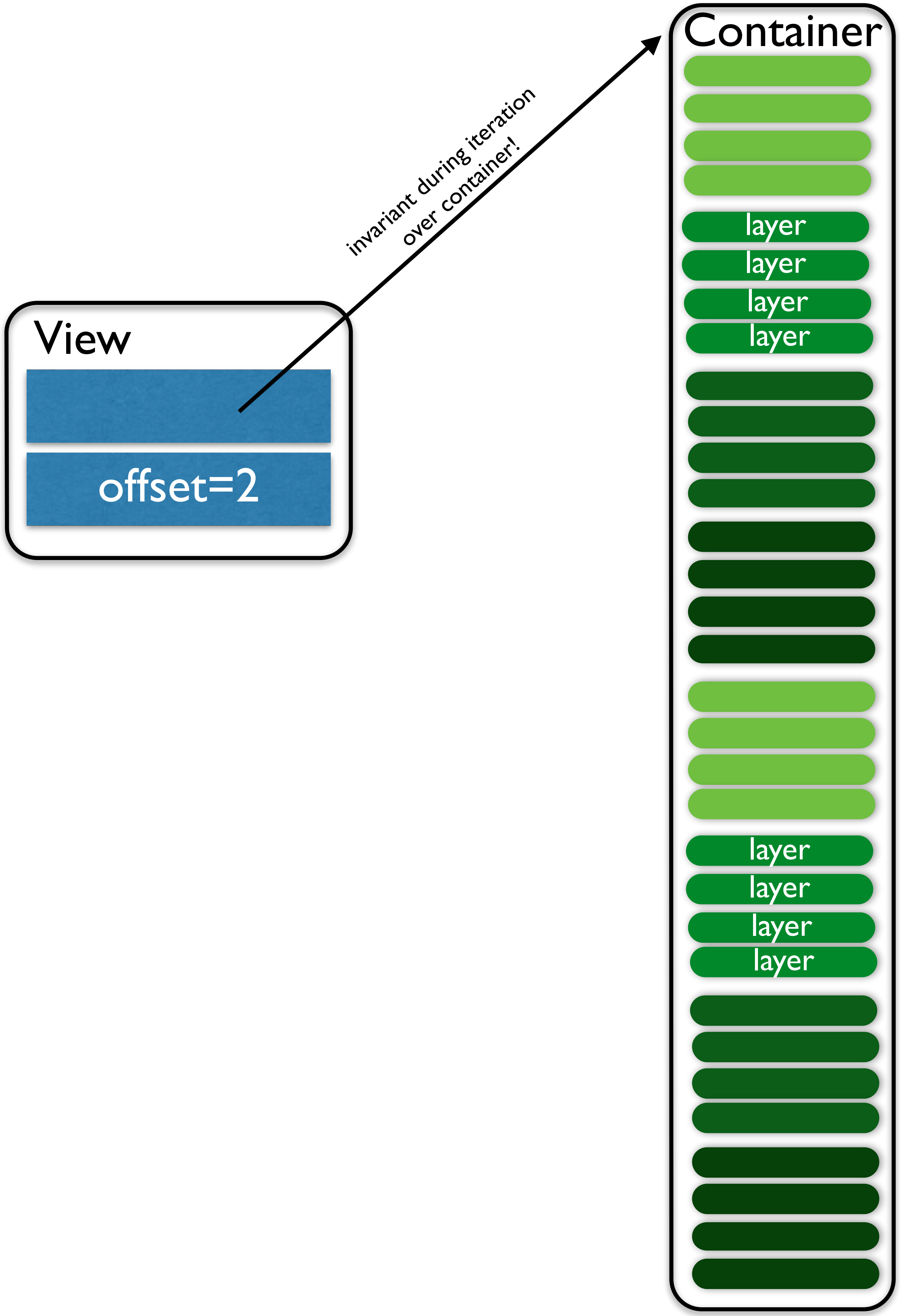
Make *collections* of objects the building block,

Data Model: *Collections* of Objects



Make *collections* of objects the building block,

Data Model: *Collections* of Objects



Try to reason about collections of objects

Do not expose the actual layout of the data

Optimal layout probably depends on actual CPU (or GPU!)

eg: optimal 'block size' probably depends on cache line size (ie. IvyBridge: 64 bytes)

Thus: Do not expose the details of this layout!!!

Borrow ideas from 'Arrow Street':

Expose a 'view' of an 'object', and iterate over 'views'

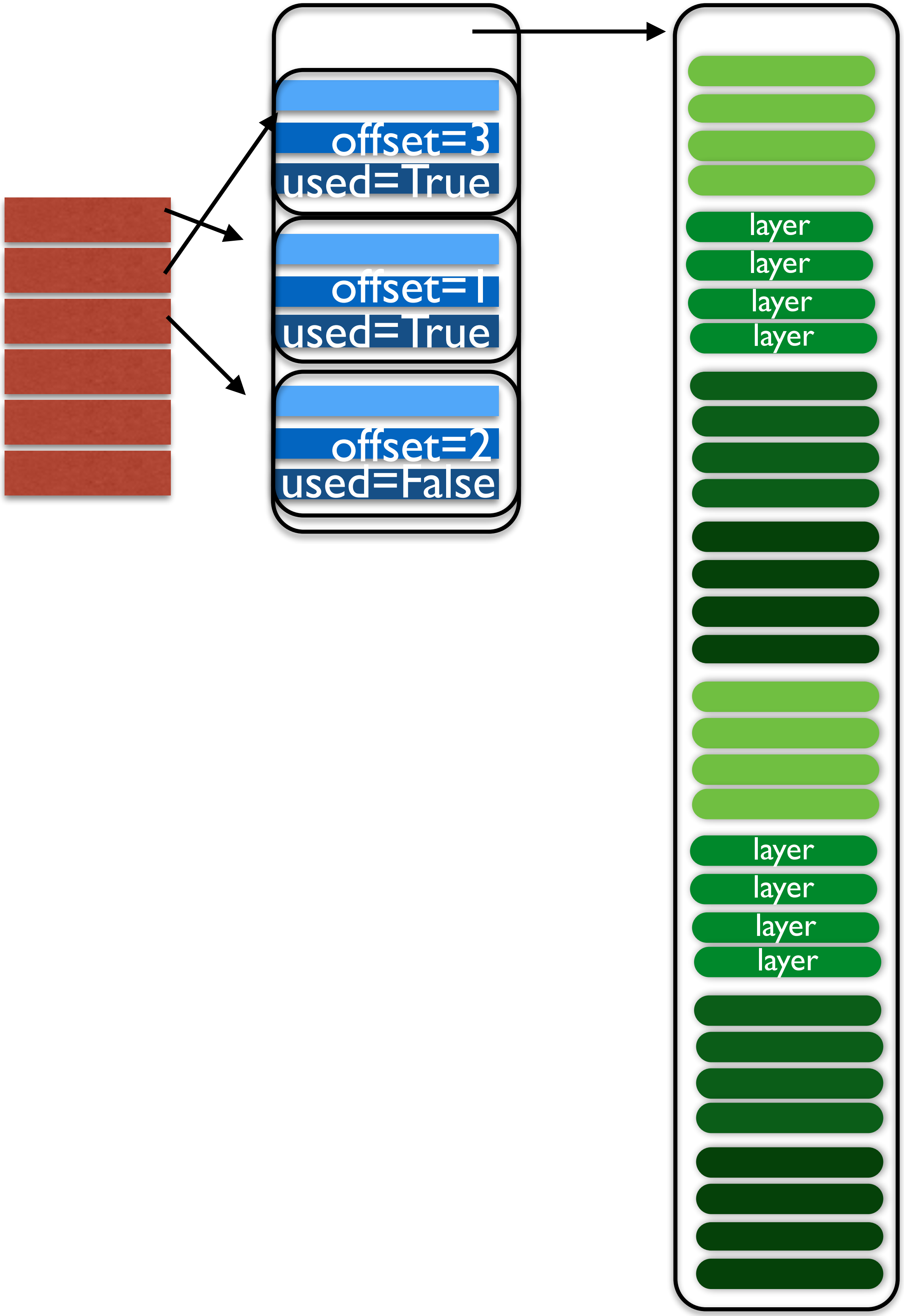
A 'view' 1) binds a container and index,

2) is obtained by dereferencing an iterator (which in turn is obtained from the container)

Effectively, a 'view' fakes an individual object, and the compiler (hopefully) optimizes it away

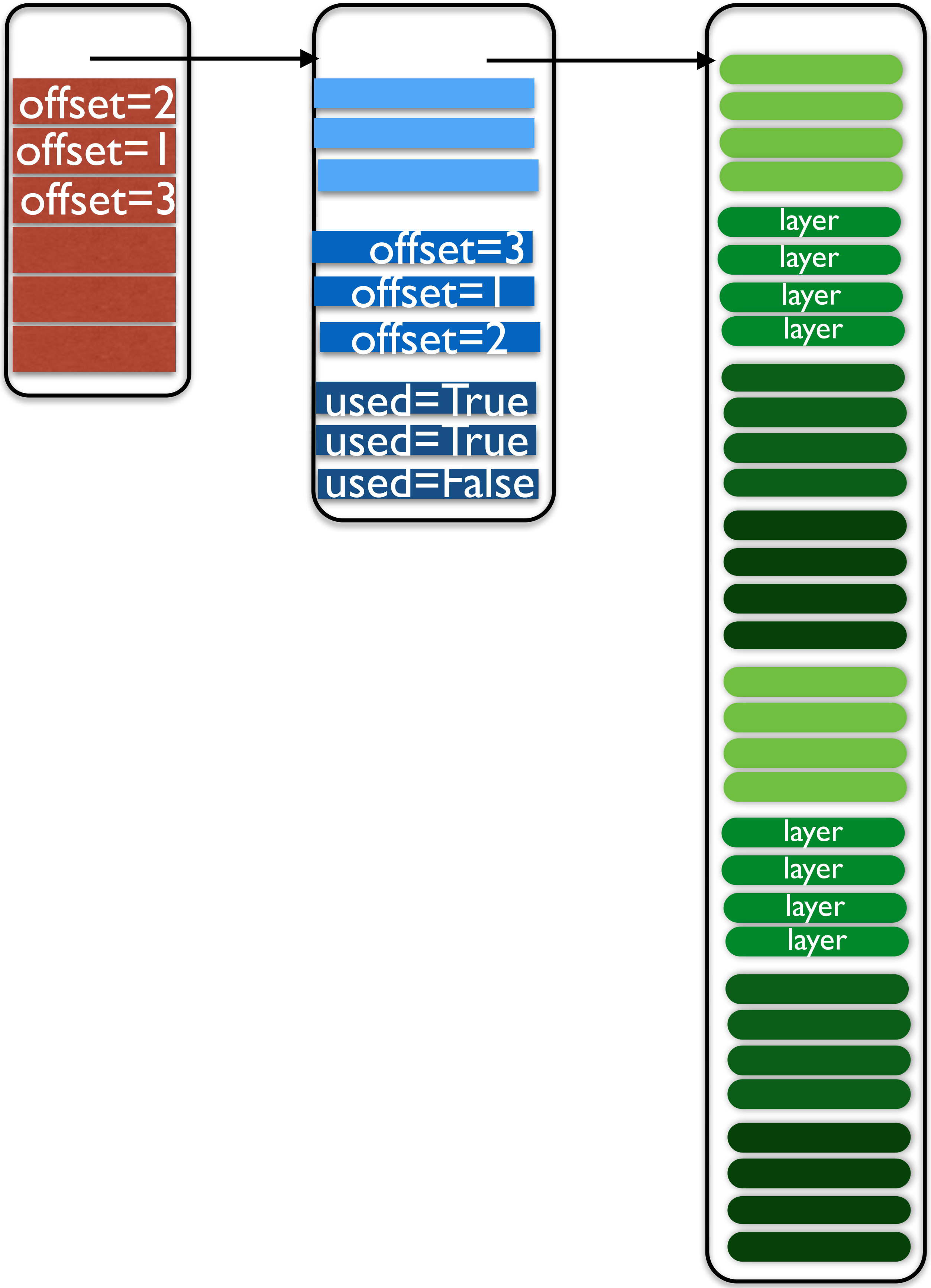
Step II: "Member-wise" data (a la 'split' mode in Root)

Data Model: *Collections* of Objects



Views could be augmented (as in eg. PatForward!)

Data Model: *Collections* of Objects



Views could be augmented (as in eg. PatForward!) by ‘nesting’

the resulting view would span multiple containers

(reminds me of an SQL ‘join’ — and ZEBRA)

Fortunately, all the details can be completely hidden!

Caution: there is an implied extra layer of indirection here (which is *bad*)

Data Model: Example Definition (one layer)

```
class HitContainer {
    private:
        // private shadow Hit_ class for (blocked) storage
        constexpr static unsigned N = 16;
        template <unsigned N_> struct Hit_ {
                std::array<double,N_> m_x;
                std::array<int,N_>    m_layer;
                std::array<int,N_>    m_flags;
        };
        std::vector<Hit_<N>> m_container;
        size_t               m_size;

        // private accessors to storage
        double x(unsigned i) const  { return m_container[i/N].m_x[i%N]; }
        int layer(unsigned i) const { return m_container[i/N].m_layer[i%N]; }
        int flags(unsigned i) const { return m_container[i/N].m_flags[i%N]; }
    public:
        class Hit ;// provide a 'view' into items in the container
        class Iterator; // provide iterator over views

        HitContainer( unsigned capacity = 0 ) ;
        void emplace_back(double x, int layer, int flags) ;

        Iterator begin() {  return Iterator(this,0);  }
        Iterator end()   {  return Iterator(this,m_size); }
};
```

```
// provider iterator over view
class HitContainer::Iterator {
    private:
        HitContainer* m_parent;
        unsigned      m_offset;
        friend HitContainer;
        Iterator(HitContainer* parent, unsigned offset);
    public:
        Hit operator*() { return Hit(m_parent,m_offset); }
        bool operator!=(const Iterator& rhs) const;
        bool operator==(const Iterator& rhs) const;
        Iterator& operator++() { ++m_offset; return *this; }
};
```

```
// provide a ‘public view' of objects in the container
class HitContainer::Hit {
    private:
        friend HitContainer;
        Hit(HitContainer* parent, unsigned offset)

        // this class ‘binds’ a container and offset….
        HitContainer* m_parent;
        unsigned          m_offset;

    public:
        // … to some public visible accessors
        double x()  const { return m_parent->x(m_offset); }
        int layer() const { return m_parent->layer(m_offset); }
        int flags() const { return m_parent->flags(m_offset); }
};
```

Note: no (explicit) pointers, no (explicit) new/delete…

Data Model: Example Usage

```
int main() {
    HitContainer c(1000);
    for (int i=0;i<1000;++i) { c.emplace_back( double(i)/100, i%4, 0 ); }
    for (const auto& hit : c ) { cout << hit.x() << " " << hit.layer() << endl; }

    double x = 0;
    for (const auto& hit : c ) {  if (hit.layer()==2) x += hit.x(); }
    cout << “sum of x for layer 2:“ x << endl;

    return 0;
}
```

in the loop :

1. ’hit’ gets elided
2. ‘hit.layer()’ and ‘hit.x()’ get fully inlined

—> HitContainer::Hit is completely optimized away…

(Apple clang-500.2.79 based on LLVM 3.3svn (OSX 10.9) at -O2)

Data Model: Proposed Next Steps

Try to implement these ideas in PatForwardHit :Tf::HitExtension<Tf::LineHit>

Critical code (major fraction of Hlt time!)

Well 'isolated' (changes to limited # of packages)

- These are NOT Event Model classes (!)

Benchmark!!!!

- the 'toy' doesn't show any real difference ;-(
- maybe (hopefully?) it is too small & too simple

If (and only if) it makes a difference, *then* consider teaching GOD to generate similar code directly from the XML description… (start with low level, eg. hits, and work upwards eventually)

Aside: ExtraInfo

Note that one could implement a 'clean' ExtraInfo this way already now.

Instead of adding a { int : double } store into each 'Particle' (with badly defined ints as keys!) add, for each 'observable', a dedicated 'table' with a single pointer to the relevant (keyed) container of particles, and [key, value]. The 'name' of this table in the TES would replace the int 'key' in ExtraInfo.

Could group 'related' observables together (i.e. value could be an object)

(note: this was proposed during the Particle event model review looooong ago)

+ : if you want to loop over eg. subset of Particles (Tracks, …) based on value of 'value': loop over 'table' — i.e. use as an 'index'.

+ : better management of what is what (use TES location for key, less change of collisions, better readable)

- : if you want, for a given Particle, to look up the 'value' — i.e. use as an ntuple

- : more work to store

One last (crazy?) suggestion

Use clang (through ROOT6 cling?) to 'just in time' compile the expressions generated by eg.

```
   (PT > 500.0*MeV)
 & (P > 5000.0*MeV)
 & (MIPCHI2DV(PRIMARY) > 4.0)
 & (((TRCHI2DOF < 2.5)& ISMUON)|(TRCHI2DOF < 2.5))
```

i.e. 'JIT' this expression in 'initialize' (and eg. run changes), use the optimized version during 'execute'

This is the only way I can (so far) think of on how inline 'composed' predicates — which is necessary for effective vectorization…

Current setup uses python as 'factory' — with the above string as the 'recipe' for what to build — to build/compose a C++ 'expression tree'; i.e. already now, after construction, there is no python running during execute.

ps. benchmarks show that this is NOT the bottleneck in CombineParticles — the fitting of vertices dominates right now.